Rational. SDL and TTCN Suite

IBM

User Manual

User´s Manual

IBM Rational SDL and TTCN Suite 6.3

User´s Manual

This edition applies to IBM Rational SDL Suite 6.3 and IBM Rational TTCN Suite 6.3 and to all subsequent releases and modifications until otherwise indicated in new editions.

Copyright Notice

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send written license inquiries to the following:

```
IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.
```

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send written inquiries to:

```
IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan
```

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions. Therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

```
Intellectual Property Dept. for Rational Software
IBM Corporation
1 Rogers Street
Cambridge, Massachusetts 02142
U.S.A.
```

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.
If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Additional legal notices are described in the legal_information.html file that is included in your software installation.

Copyright License

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_.

Trademarks

See http://www.ibm.com/legal/copytrade.html.

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.html.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

Chapter

1 User Interface and Basic Operations

This chapter describes the general graphical concepts that are used throughout the user interfaces of the SDL Suite and TTCN Suite tools. It also describes some common menu choices. This information will not be repeated in other chapters of the User's Manual.

You should have a basic understanding of the SDL Suite and TTCN Suite concepts and tool family before you read this chapter. Such information can be found in:

- ***chapter 2, Introduction to the SDL Suite, in the SDL Suite 6.2 Getting Started***
- **chapter 3, *Introduction to the TTCN Suite (on UNIX), in the TTCN Suite 6.2 Getting Started***
- **chapter 2, *Introduction to the TTCN Suite (in Windows), in the TTCN Suite 6.2 Getting Started***

Graphical Environments and Style Guides

The SDL Suite and the TTCN Suite tools family is designed to be available on workstations running UNIX and on PCs running Windows. Since the majority of the tools are graphical, it is assumed that the computer you are to run the tools on has the required graphical support.

IBM Rational supports the following graphical environments:

- On UNIX workstations, the X Windows system, managed by the Motif window manager. The versions currently supported are X11 R6 and Motif 2.1.
- On PCs, the Microsoft Windows system. SDL Suite and TTCN Suite supports Windows 2000 and Windows XP.

Throughout the manuals, it is assumed that you are familiar with the graphical environment that is currently used. Otherwise, we recommend you to read the literature that describes that graphical environment. See for instance "References" on page 38.

The SDL Suite for Windows is fully compatible with the SDL Suite on UNIX workstations, with respect to functionality and storage formats. However, since they are implemented on different graphical environments, there may be slight differences in the appearance of each tool. Also, we have tried to adopt the respective style guide for each environment as long as feasible, but have been forced to compromise in order to provide a uniform user interface between the UNIX workstation and PC environments.

Constraints

The constraints that are imposed on SDL Suite and TTCN Suite by the graphical environment are identified in "Microsoft Windows System Factors" on page 46, "X Window System Factors (UNIX only)" on page 47 and "OSF/Motif Factors (UNIX only)" on page 49 in chapter 4, *System Setup, in the Installation Guide*.

Application Windows

All SDL Suite and TTCN Suite tools have a main window with a common general appearance. The tools may also have sub windows that depend on the main window. Apart from these, there are also dialog windows, see “Dialog Windows” on page 28.

In the TTCN Suite **in Windows**, the windows will be displayed in the working area of the TTCN Suite desktop.

Main Windows

A general main window of an SDL Suite and TTCN Suite tool in Windows and on UNIX, as well as the TTCN Suite desktop in Windows, is depicted and explained below:

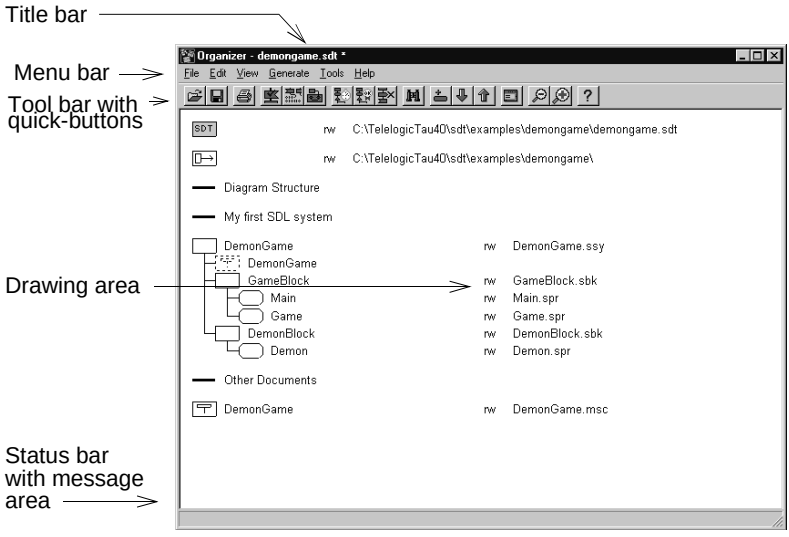


Figure 1: A main window **(in Windows)**

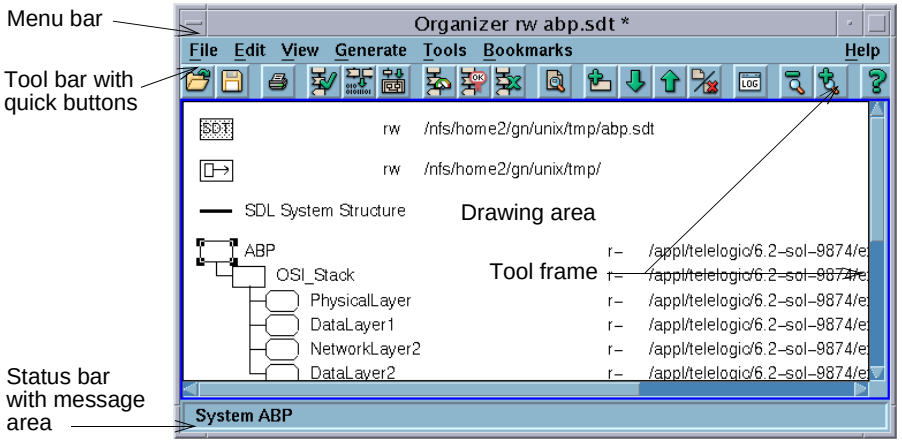


Figure 2: A main window **(on UNIX)**

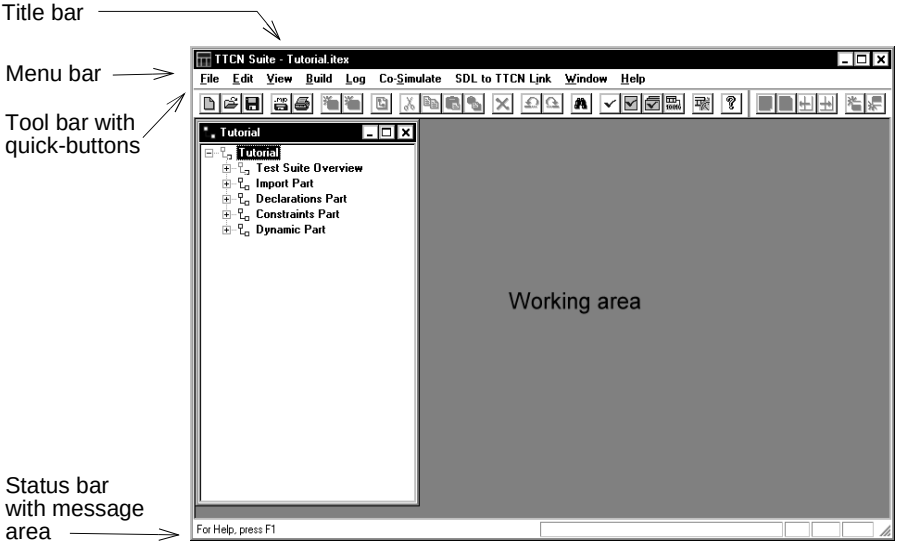


Figure 3: The TTCN Suite desktop **(in Windows)**

Title Bar

The title bar may identify the tool family, the tool and the opened document. In some tools, an asterisk signifies that the document is not saved since the last change.

Menu Bar

The menu bar contains pull-down menus. The menu choices operate either on the whole document or file that is opened in the tool, or on any selected object(s). More information on menus and menu choices can be found in "General Menus" on page 8 and "General Menu Choices" on page 8.

You can "preview" the functionality of a menu choice by pointing to it. In the SDL Suite **on UNIX** you also have to press the mouse button. An explanation will appear in the status bar.

If a menu or menu choice is dimmed, it is not appropriate or meaningful in the current situation, or the associated tool license is temporarily lost.

If a menu or menu choice is hidden, you do not have a license for the associated tool, or the tool has a long/short menu concept which allows hiding of menu choices.

Tool Bar

Most operations that the quick-buttons in the tool bar provide, have an exactly corresponding menu choice, but some quick-buttons have a slightly different functionality compared to their corresponding menu choice. See also "General Quick-Buttons" on page 24.

You can "preview" the functionality of a quick button by pointing to it. In the TTCN Suite **on UNIX** you also have to press the mouse button. An explanation will appear in the status bar, and as a tool tip just below the button (not available in the TTCN Suite **on UNIX**).

In the TTCN Suite **in Windows**, the tool bars are dockable on all sides of the desktop.

Drawing Area

The information a tool handles is displayed and may possibly be edited in the drawing area. The drawing area may also give access to popup menus. See also "The Drawing Area" on page 25.

Working Area (only in the TTCN Suite in Windows)

The working area of the desktop will display the TTCN Suite windows that you open. For example, when you open a TTCN document, the Browser window will be displayed in the working area.

Tool Frame (only in the SDL Suite on UNIX)

A tool frame, with a unique color for each tool, surrounds the drawing areas in the SDL Suite. This makes it easier for you to identify a tool, without the need to see the whole drawing area or the title bar.

Status Bar

The status bar displays information about menu choices and quick-buttons, the progress or result of an invoked operation, and the name of a selected object or additional information about an object.

Sub Windows

A sub window of an SDL Suite and TTCN Suite tool is a window that depends on the main window of the tool. A sub window is opened either automatically when the main window is opened or as a result of user interaction with the main window. All sub windows are automatically closed when the main window exits. However, to close a sub window does not affect the main window.

The TTCN Suite Logs

Some TTCN Suite tools, for example the Analyzer, produce logs.

On UNIX, the relevant tool dialog allows the log device to be set to *Screen*, *File* or *None*. If set to *Screen*, the log will be displayed in its own log window. If set to *File*, a directory dialog will be displayed where you can specify a file to save the log in.

In Windows, the Log Manager will display the logs, and tabs are used for viewing different logs. The contents of the window can be copied and saved. For more information, see "Viewing Log Information" on page 1279 in chapter 30, *Editing TTCN Documents (in Windows)*.

Zooming a Window

You can change the scale of the drawing area of an SDL Suite window by selecting *Set Scale* in the *View* menu or by using quick buttons for zooming in and out. The scale is normally between 20% and 800%.

General Menus

The following menus are generally available in a menu bar:

- The *File* menu contains menu choices related to files, documents and the whole drawing area, as well as the tool/window itself. Examples: *New*, *Open*, *Save*, *Print* and *Exit*.
- The *Edit* menu contains menu choices concerning editing of the current document. It may also contain menu choices for changing objects. Examples: *Undo*, *Cut*, *Copy*, *Paste*, *Add* and *Edit.*
- The *View* menu contains menu choices that changes the appearance of the window and the information in the drawing area. Examples: *Expand*, *Collapse, Window Options* and *Set Scale*.
- The *Tools* menu contains menu choices for starting tools and utilities, and opening or raising other windows. Examples: *Search, Show Organizer* and *Show <window name>*.
- The *Help* menu contains menu choices for supporting you with help and other useful information.

General Menu Choices

Some common menu choices are described below and the descriptions will not be repeated elsewhere.

File Menu

The most common menu choices in the *File* menu are only described in this section. If a tool provides additional menu choices in the *File* menu, they will be described in the corresponding chapter. If a common menu choice has a different functionality, it will also be described in its corresponding chapter but minor differences will be mentioned below.

New

When you select this menu choice, a new document, diagram or file will be created and displayed. In some tools, it will be given a default file name which you later may change, and in some tools you have to specify the name and the type of document/diagram in a dialog before it will be displayed.

- For information on adding new pages to SDL diagrams, see "Add" on page 2035 in chapter 43, *Using the SDL Editor*.
- In the Text Editor, you have to select in a dialog whether to create a new file, copy an existing file, or to copy a template file. The directory for template files is defined by the preference TE*TextTemplateDirectory. For more information, see "Text Editor Preferences" on page 266 in chapter 3, *The Preference Manager*.
- When you create a new link file in the Link Manager, the current endpoint and link information will be deleted. You will be warned if a link file is already opened.

Open

When you select this menu choice, a file selection dialog will be issued, in which you select what file to open. For more information, see "File Selection Dialog" on page 30.

A default file filter, which depends on the tool, will be used. It corresponds to the default file name extensions applied when a document or diagram is saved (see "Save" on page 11). The tools and the filters are listed in the table below.

- When you open a new SDL simulator, a currently running simulator will be stopped after user confirmation. The Watch window will be updated and the text area will be cleared from previously executed commands.
- When you open a new link file in the Link Manager, a currently opened and modified file has to be saved first.

Tool	Default File Filter
SDL Target Tester	`*.sym`
SDL Coverage Viewer	`*.cov`
HMSC Editor	`*.mrm`
Index Viewer	`*.xrf`
The TTCN Suite **in Windows**	`*.itex`
Link Manager	`*.sli`
MSC Editor	`*.m??`
OM Editor	`*.som`
Organizer Log	`*.log`
SDL Editor	`*.s??`
SDL Simulator	`*_sm*.exe` **(in Windows)** `*.sct` **(on UNIX)**
SDL Explorer	`*_vl*.exe` **(in Windows)** `*.val` **(on UNIX)**
State Chart Editor	`*.ssc`
Text Editor	`*.txt`

Note:

No specific file name extension on diagrams are imposed. If other file name extensions than the ones suggested have been used when saving diagrams, a different filter than the given must be applied.

Note: The MSC Editor and instance oriented MSC/PR

The MSC Editor supports reading MSC/GR (default file name extension is `.msc`) and MSC/PR (default file name extension is `.mpr`). However, the MSC Editor cannot read MSC/PR expressed in instance-oriented form. Only the event-oriented form can be read by the tool. See the Z.120 recommendation for more information on these two alternative formats.

Note: Opening *.cif files in SDL editor

If a *.cif file is opened in an SDL editor, then the information from the SDL/CIF file is presented in a new, dirty and unsaved diagram buffer.

Save

When you select this menu choice, the current information in the document/diagram/window will be saved in a file.

When you save a document/diagram for the first time, a file selection dialog is opened. A default file name and file name extension is suggested. You have to specify where the file is to be saved and you may also change the file name and extension. If you try to overwrite an existing file, you will be warned.

Note:

If you keep the file name extensions for different documents/diagrams listed below, it will be easier to locate the files. It is also impossible to add an existing file to the Organizer unless it has the default extension.

Document/Diagram Type or Storage Format	Default File or Extension
Command definitions (SDL Simulator UI)	`.cmds`
Coverage	`.cov`
HMSC	`.mrm`
Link file	`.sli`
MSC	`.msc`
Object Model	`.som`
Organizer Log	`.log`
SDL System	`.ssy`
SDL Block	`.sbk`
SDL Substructure	`.ssu`
SDL Service	`.ssv`

Document/Diagram Type or Storage Format	Default File or Extension
SDL Process	.spr
SDL Procedure	.spd
SDL System Type	.sst
SDL Block Type	.sbt
SDL Service Type	.svt
SDL Process Type	.spt
SDL Macro Definition	.smc
SDL Operator	.sop
SDL Package	.sun
SDL Overview	.sov
State Chart	.ssc
State Overview	.ins
Text	.txt
TTCN test suite **(in Windows)**	.itex .mp .imp
Variable definitions (SDL Simulator UI)	.vars

Save Document

The *Save Document* menu choice only exists in the TTCN Browser and Table Editor **on UNIX**. When you select it, the TTCN document will be saved. See also "Save" on page 11.

Save As

Select this menu choice to save the current document/diagram in a new file. A default file name and file name extension will be suggested, see "Save" on page 11. If you change the file name and extension and try to overwrite an existing file, you will be warned.

If the document/diagram was already connected in the Organizer, it will be reconnected to the newly created file. This does not happen in the following cases:

- *Save As Text*. By specifying *.txt as the file name extension for graphical diagrams, the diagram text will be saved in the specified file.
- *Save As CIF*. By specifying *.cif as the file name extension for SDL diagrams, the diagram will be saved in SDL/CIF format, a standardized textual format that includes layout information. When this extension is used in the SDL editor and the Save As dialog is closed, you will be asked to select diagram pages to include in the generated file. As default, all pages are selected. A *.cif file can be opened in the SDL editor. When doing so, the information from the *.cif file is presented as a new, dirty and unsaved diagram buffer.

Note:

If you keep the file name extensions for different documents/diagrams listed above, it will be easier to locate the files. It is also impossible to add an existing file to the Organizer unless it has the default extension. In the TTCN Suite **in Windows**, you can only open test suites that have the extension `.itex`, `.mp` or `.imp`.

- In the TTCN Suite **on UNIX**, there are three alternatives for compression when you save as TTCN-GR: *None*, *Gzip* and *Compress*. *None* gives maximum compatibility. *Gzip* is preferred over *Compress*, both for portability and performance.

Save a Copy As

This menu choice is available in the SDL Suite editors. It saves a copy of the current document/diagram in a new file.

Note:

The window will hold the **original** file, **not the newly saved copy**. The document/diagram remains connected to the old file and the Organizer's structure is left unaffected by the operation.

The name of the new file is to be specified in a file selection dialog and a default file name and extension will be suggested; see "Save" on page 11. If you try to overwrite an existing file, you will be warned.

Save All

This menu choice is available in the SDL Suite editors. It saves all modified documents/diagrams that are opened. Except for that, it works as described in "Save" on page 11.

Print

The different *Print* dialogs and how to print is described in chapter 5, *Printing Documents and Diagrams*.

Close

When you select this menu choice, the current document/diagram (or in some cases the window) will be closed. In most documents, you will not have to confirm the closing.

- In the TTCN Suite, when you close the last open Browser on a document, you will be asked for confirmation if it has not been saved.
- When you close the Link Manager window, the Link Manager will still be active in the background but the user interface will not be available.
- In the SDL and TTCN Integrated Simulator Editor **on UNIX**, the window is closed, but current document is not closed until the associated setup is closed.
- In the SimUI's Command window, the commands currently displayed will be saved and shown again when the window is opened again. This is also valid for the variables in the Watch window.

Close Diagram

This menu choice is available in the SDL Suite editors. When you select it, the current diagram will be closed. If changes in the diagram have not been saved, a dialog will be issued where you can select to save or not before closing.

If the closed diagram was the last open diagram, the editor will exit. If there are more editor windows open, the window will be closed. If there is only one editor window, but more diagrams open, another diagram will be displayed in the window.

Revert Diagram

This menu choice is available in the SDL Suite editors. When you select it, the current diagram will be re-loaded from the file system. This operation is useful if the file has changed in any way in the file system.

Exit

When you select this menu choice, the current tool will exit. If a document/diagram has been changed since last save, a dialog will be issued where you may select to save it, not save it or – where applicable – save all opened documents/diagrams or quit without saving anything.

- In the SDL Simulator, you will be asked to save unsaved changes to the SimUI's definition files. See "Definition Files" on page 2221 in chapter 49, *The SDL Simulator*.

Tools Menu

A *Tools* menu is available in most tools. It contains various menu choices, but the following is included in almost all *Tools* menus:

Show Organizer

This menu choice raises the parent Organizer main window, that is from where the tool was started.

Help Menu

A *Help* menu is available in most main and sub windows of the tools. However, the menu choices in this menu are not the same in each tool or on each platform, but they work in the same way: When you select a menu choice in the *Help* menu, a help viewer will be opened with the corresponding help topic.

The following *Help* menu choices, that may need an explanation, are described below:

- *About <Tool>*
- *Help Desk*
- *Index*
- *Latest News*
- *License Information*
- *License Info*

- *New Features*
- *On Field*
- *On Shortcuts*
- *On Window*
- *Search*
- *IBM Home Page*

About <Tool>

Opens the *About* message box for the tool that the menu choice was invoked from, which gives information on the current tool version, copyright, etc.

Help Desk

Opens the online help to guide the user how to contact IBM Rational Software Support, http://www.ibm.com/software/awdtools/sdlsuite/support or http://www.ibm.com/software/awdtools/ttcnsuite/support, see also "How to Contact Customer Support" on page *iv in the Release Guide*. This menu choice is only available in the Organizer.

Index

Opens the help viewer with an index of all entries in the documentation. This menu choice is only available in the Organizer.

Latest News

Opens the help viewer with a "readme" file, containing late information about the release.

License Information

Opens a dialog with license information for all tools. For more information on this dialog, see "License Information" on page 166 in chapter 2, *The Organizer*. This menu choice is available in the Organizer.

License Info

On UNIX, this menu choice opens a dialog with additional license information on the TTCN Suite tools. For more information, see "License Info" on page 1122 in chapter 24, *The TTCN Browser (on UNIX)*. **In Win-**

dows, the dialog lists the IBM Rational licenses currently in use as well as the number of licenses available for each tool.

New Features

Opens the help viewer with information about new and changed functionality.

On Field

This menu choice is available in the TTCN Table Editor **on UNIX**. It displays the TTCN BNF syntax applicable for the relevant field.

On Shortcuts

Opens the help viewer with information about various user interface shortcuts that can be used within the Organizer tool.

On Window

Opens a text describing the TTCN Suite tool that the menu choice was invoked from. This menu choice is only available in the TTCN Suite **on UNIX**.

Search

Starts a textual search across all help files. This menu choice is available in the Organizer and all the SDL Suite tools.

IBM Home Page

Opens the web browser with IBM's home page on the World Wide Web (`http://www.ibm.com/software/rational`). This menu choice is only available in the Organizer.

Defining Menus in the SDL Suite

The graphical the SDL Suite tools allow you to define additional menus and menu items that execute external commands or send PostMaster messages. Separate user-defined menus can be defined and added to each graphical tool. However, the pre-defined menus in each tool can **not** be removed or changed.

User-defined menus are described by menu definition files that are read by the tools when they start up.

It is sometimes more useful to add user-defined menus to the tools after they have been started; this can be accomplished if you use the IBM Rational Public Interface. For more information see "Menu Manipulation Services" on page 525 in chapter 11, *The Public Interface*.

> **Note:**
>
> The fact that a set of menus have been defined in a menu definition file, does not prevent using the public interface services to add additional menus, or even to modify the menus that were defined by the menu definition file.

Tools and Menu Definition File Names

Menu definition files must have fixed names that indicate which tool the menus are intended for.

The following tools allow addition of user-defined menus, and also offer the possibility of letting the menu commands access the internal state information of the tool. This is accomplished if you use *format codes* that are documented separately for each tool in the Public Interface.

Tool	Menu definition file name
Organizer	`org-menus.ini`
MSC Editor	`msce-menus.ini`
SDL Editor	`sdle-menus.ini`
OM/SC/HMSC Editor	`ome-menus.ini`
Text Editor	`te-menus.ini`

The following tools allow addition of user-defined menus, but do not offer the possibility of accessing the internal state of the tool.

Tool	Menu definition file name
Preference Manager	`pref-menus.ini`
Type Viewer	`typ-menus.ini`
Coverage Viewer	`cover-menus.ini`
Index Viewer	`xref-menus.ini`
Tree Viewer	`tree-menus.ini`
Simulator/Explorer UI [a]	`simui-menus.ini`

a. **In Windows**, the Simulator/Explorer UI's can **not** read menu-definition files.

Menu Definition File Location

On start-up, each tool that supports menu-definition files will search for a menu definition file with the given pre-defined name in up to three locations:

1. First, the directory from where the SDL Suite was started is searched. If a file with the expected name is found, the tool will attempt to read it and install the menus described therein.
2. Second, the directory named in the `HOME` environment variable will be searched. If a file with the expected name is found, it will be used.
3. Last, the directory where the SDL Suite is installed will be searched (**on UNIX**, `$telelogic/`, and **in Windows** the top installation directory, by default `C:\IBM\Rational\SDL_TTCN_Suite6.3`).

If no file is found, no user-defined menus will be added on start-up.

Format of Menu Definition Files

A menu definition file is a line-oriented text file separated into sections by lines containing special section markers. Each section contains lines formatted in the same way, containing an option/value pair. Each section describes either a menu or a menu item.

The first line of a menu definition file is a format tag that identifies the file as menu definition file:

```
SDT-DYNAMICMENUS-6.3
```

Each section is started by adding a line with a section name between brackets "[ ]". Valid sections in a menu definition file are:

```
[MENU]
[MENUITEM]
[MENUEND]
```

[MENU] Section

The menu section starts a new menu. Subsequent `[MENUITEM]` sections will add a menu item to this menu until a `[MENUEND]` section is encountered. A `[MENU]` section should be the first line after the initial tag or follow directly after a `[MENUEND]` section.

After the `[MENU]` section tag follows the option below, using the syntax:

`Name=`*`NameOfMenu`*

Option	Explanation/Value
`Name`	A string that contains the name of the menu. The name is presented in the tool's menu bar. The ampersand '&' character may be placed just in front of a letter to indicate that this letter will be underlined in the menu name and thus function as a keyboard shortcut for menu traversal. Make sure that the letter is not used as a shortcut in any other menu in the menu bar, or it may not be possible to open the menu with the keyboard.

[MENUITEM] Section

A `[MENUITEM]` section must occur between a `[MENU]` and a `[MENUEND]` section.

Following the `[MENUITEM]` section tag is a number of options and their values, using the syntax:

```
Option=Value
```

This section adds a menu choice to the specified menu. The menu choice could either perform an OS command or issue a PostMaster notification when selected. The OS command to perform or the message to broadcast could be sensitive on a selected symbol.

The exact interpretation of two of these options (`ProprietaryKey` and `AttributeKey`, described below) will depend on which tool the menu will be installed in. In particular, the `ProprietaryKey` option will only have significance in the Organizer and the graphical editors. The `AttributeKey` option will only have significance in the graphical editors. If not used these options should be set to 0.

For tools supporting access to internal state information, format codes can be used in the command string or as message parameter, providing additional context-sensitive information.

For more information on options and format codes, see *"Add Item to Menu" on page 584 in chapter 11, The Public Interface*.

Option	Explanation/Value
`ItemName`	A string that contains the menu item text that appears in the menu item. Ampersand syntax is supported, as for menu names. Make sure that different letters are selected for each menu item in the menu; otherwise keyboard activation of the menu item may not work.
`Separator`	A boolean value (0 or 1) that indicates whether a separator (a thin line) should precede the menu item in the menu.
`StatusbarText`	A string that should be displayed in the tool's status bar while the menu item is selected to hint you about the function of the menu choice.

Option	Explanation/Value
`ProprietaryKey`	An integer whose interpretation depends on the tool; the Organizer interprets this parameter as `lastAction`, and the graphical editors interpret it as `ProprietaryKey`. For more information, see "lastAction" on page 589 and "ProprietaryKey" on page 594 in chapter 11, *The Public Interface*. For the Organizer, you can use either an integer or a symbolic string as the value. If not used, simply set to 0.
`AttributeKey`	An integer whose interpretation depends on the tool; the graphical editors interpret this parameter as `AttributeKey`. For more information, see "AttributeKey" on page 594 in chapter 11, *The Public Interface*. If not used, simply set to 0.
`Scope`	An enumerated value that indicates when the menu item should be dimmed. For the valid values, see "scope" on page 585 in chapter 11, *The Public Interface*. You can use either an integer or a symbolic string as the value. If not used, simply set to 0.
`ConfirmText`	A text string that contains a dialog box text. If not empty, this will issue a two button dialog with *OK* and *Cancel* buttons and an editable text field containing the command to be performed. You can alter the command text before pressing *OK*.
`ActionInterpretation`	An integer value indicating the desired action when a menu item is activated: `PM_MESSAGE` (0) - send PostMaster message `OS_COMMAND` (1) - execute OS command
`BlockCommand`	Only significant if `Action` is set to 1: A boolean value (0 or 1) indicating whether the Organizer should wait for the execution of the command to complete (1), or allow you to perform other operations while the command executes (0).

Option	Explanation/Value
FormattedCommand	Only significant if Action is set to 1: The OS command to perform. Some tools evaluate specific context sensitive format codes.
MessageNumber	Only significant if Action is set to 0: Indicates the number of the PostMaster message to send.
FormattedMessage	Only significant if Action is set to 0: The parameters to the PostMaster message. Some tools evaluate specific context sensitive format codes.

[MENUEND] Section

The [MENUEND] section indicates that a menu definition has come to an end and that no more [MENUITEM] sections should appear until a new [MENU] section is encountered. This section has no options.

Example of a Menu Definition File

An example of a typical menu-definition file could be:

```
SDT-DYNAMICMENUS-3.6
[MENU]
Name=&RCS
[MENUITEM]
ItemName=Check &Out
Separator=0
StatusbarText=Check out the selected object
ProprietaryKey=1
AttributeKey=0
Scope=SELECTED_OBJECT_NOT_IN_EDITOR
ConfirmText=
ActionInterpretation=OS_COMMAND
BlockCommand=1
FormattedCommand=co %f
[MENUEND]
```

General Quick-Buttons

If you hear a beep when you click a quick-button, then the operation is currently not available.

The following quick-buttons can be considered as standard buttons that are available in many tools:

Back
The same as *Back* in the *Diagrams* menu.

Forward
The same as *Forward* in the *Diagrams* menu.

Open
The same as *Open* in the *File* menu.

Save
The same as *Save* in the *File* menu.

Print
The same as *Print* in the *File* menu.

Search
The same as *Search* in the *Tools* menu.

Search Again
The same as *Search Again* in the *Tools* menu.

Show Organizer
Raises the parent Organizer window, from where the tool was started. In a sub window to a tool, this button raises the main window of the tool.

Decrease Scale
Decrease the scale of the drawing area by 20%.

Increase Scale
Increase the scale of the drawing area by 20%.

Help
The same as *On <Tool>* in the *Help* menu.

The Drawing Area

Popup Menus

You invoke a popup menu with the right mouse button. A menu choice in a popup menu generally have the same name and work in the same way as a corresponding menu choice in the menu bar.

There are two types of popup menus: context sensitive popup menus and the background popup menu.

Context Sensitive Popup Menus

Objects visible in the drawing area may have associated popup menus, whose contents may vary with the type of the object.

In the SDL Suite, a context sensitive popup menu can be displayed in two slightly different ways:

- By a right-click directly on a visible object.

  The current selection also changes with this operation, i.e. the object pointed at becomes selected and replaces any previous selection.

- By a right-click on the background, outside any object, when there exists a current selection.

  In this case, the popup menu applies to the selected object(s) and the selection does not change.

In the TTCN Suite, the popup menu **always** applies to the object the mouse pointer is placed on, but the current selection **does not change**. If you right-click on the background of the drawing area, outside of any visible object, the background popup menu is displayed, regardless of any current selection.

The Background Popup Menu

In the SDL Suite, the background popup menu is displayed if you right-click on the drawing area background, and there is no current selection in the drawing area. This popup menu contains some of the most important and/or commonly performed menu choices from the menu bar. The popup menu choices operate on the whole file, document or drawing area, in the same way as the corresponding menu choices from the menu bar do.

In the TTCN Suite, the background popup menu is always displayed if you right-click on the drawing area background, regardless of any selected objects.

List and Tree Structures

Many tools can present a hierarchical structure in two ways; as a list structure and as a tree structure. The example below is from the Organizer.

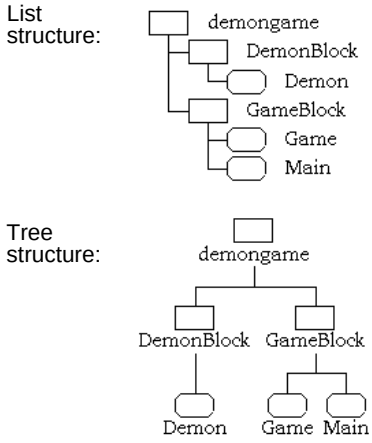


Figure 4: List and tree structures

It is possible to switch between the two structures if you select an *Option* menu choice on the *View* menu. It is also possible to collapse and expand nodes in a hierarchical structure with the *Expand* and *Collapse* commands in the *View* menu. A collapsed node is indicated by a small triangle below the node:

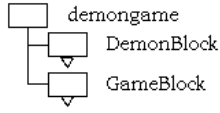


Figure 5: Collapsed nodes in list structure

Selections and Input Focus in the TTCN Suite (on UNIX)

In the TTCN Suite **on UNIX** there is a difference between *selection* and *input focus*. Selection is a marked item or text string. You select items with the left mouse button. The input focus is set when you point to an item and click with the middle mouse button. The input focus is indicated by a rectangular border surrounding the relevant item.

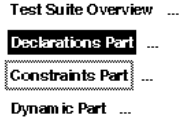


Figure 6: Selection and input focus in the TTCN Suite **on UNIX**

Dialog Windows

There are two types of dialogs, *modeless* and *modal*. A modeless dialog does not prevent you from using other parts of the application while the dialog is visible. A modal dialog does prevent this, that is, the dialog must be dealt with before any work can continue.

Modeless Dialogs

A modeless dialog can be considered as an extension to the main window. It contains an *Apply* button which executes the functionality of the dialog but does not close it. The button is often renamed to indicate the functionality, for example *Search*. The *Close* button closes the dialog but does not apply any functionality.

Note:

There is no button with the functionality of an *OK* button, i.e. execute the functionality **and** close down the dialog.

Modal Dialogs

A modal dialog is used for operations that must be confirmed or that affect the view or the information used in the tool. A typical example is an *Open* dialog. The *OK* button in a modal dialog both executes the functionality and closes the dialog. The *Cancel* button closes the dialog but the settings you may have changed in the dialog are ignored.

File Name Completion

Wherever you are supposed to input a file or directory name as text in a text field in a dialog, you can take advantage of file name completion. To do this, you type the beginning of an absolute file name or a directory and then press `<Space>`. This will add characters at the end of the text field. If there are several matches, the initial characters of the matching names will be added. Then you have to press `<Space>` again to get the alternatives one by one. And after the final alternative, you will get a space character.

File name completion is not provided in the TTCN Suite.

Example 1: File name completion

Suppose you have the files `/home/lat/hello.txt` and `/home/lat/henderson.txt`.

1. Type **`/home/l`**, and then press `<Space>`.
   The result will be `/home/lat/`.
2. Then type an additional **`h`**, and then press `<Space>` again.
   The result will be `/home/lat/he`.
3. Press `<Space>` again.
   The result will be `/home/lat/hello.txt`.
4. Press `<Space>` again.
   The result will be `/home/lat/henderson.txt`.
5. Press `<Space>` again.
   The result will be `/home/lat/he` (with a space character at the end).

Folder Button

When the name of a file or directory is to be specified in a dialog, a combination of a text field and a *folder button* is often used. You can type the name directly in the text field, or press the folder button to open a file or directory selection dialog (see the following subsections).

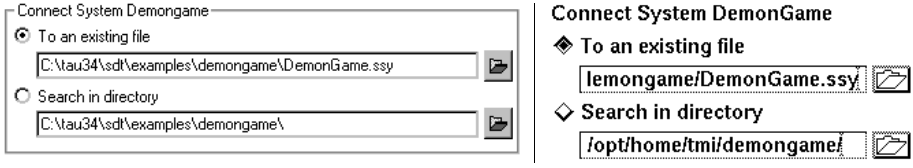


Figure 7: Folder buttons for selecting files and directories (Windows and UNIX)

File Selection Dialog

When you are going to save or open (or something similar) a file, a file selection dialog will be issued. The dialog will also be opened if you click on the folder button associated with a file name field in a dialog.

In Windows, the file selection dialog looks and behaves exactly like the normal file selection dialog used in Microsoft Windows.

In the SDL Suite **on UNIX**, the file selection dialog behaves like described below. The the TTCN Suite version **on UNIX** is described in "The TTCN Suite File and Directory Dialog (on UNIX)" on page 31.

- The text in the *Filter* field determines which files are shown in the *Files* list. The field is initially set to match the file type that the invoking tool or dialog operates on but you may change it.
- If you click the *Filter* button, the file matching pattern in the *Filter* text field is applied to the current directory in the *Directories* list and the *Files* list is updated. This is the same as pressing `<Return>` or clicking the default button when no file name is present in the *File* text field.
- If you click the *Current* button, the directory will be changed to the source or target directory, depending on which kind of operation you have initiated. For more information, see "Set Directories" on page 71 in chapter 2, *The Organizer*. The *Files* list will be updated accordingly.
- The *Directories* list shows the absolute path to the currently chosen directory, as well as the names of any subdirectories in that directory. The directories in the path are shown within brackets, slightly indented for each directory level. The last directory in the list is the name of the currently chosen directory. To change to another directory, double-click on any directory in the list. Both the *Directories* list and the *Files* list are then updated to show the contents of the new directory.
- The *Files* list shows the files in the currently chosen directory that match the pattern in the *Filter* text field. You can select the file to operate on from this list, in which case the *File* text field is updated to contain the name of the file. To double-click on a file name is the same as selecting it and then pressing the default button.

- The *File* text field contains the name of the file selected in the *Files* list. This field may initially contain a default file name, suggested by the invoking tool or dialog. You can enter any file name in this field, including a relative or absolute path. You may also specify paths to home directories with the "tilde" syntax (`~/` or `~user/`).

  When `<Return>` or the default button is pressed, it is the file in this text field that is operated on. If the field is empty and no file is selected in the *Files* list, this is the same as pressing the *Filter* button.

Directory Selection Dialog

A directory selection dialog is opened when you select menu choices dealing with directories. The dialog will also be opened if you click on the folder button associated with a directory name field in a dialog.

In Windows, the directory selection dialog looks and behaves exactly like the normal directory selection dialog used in Microsoft Windows.

In the SDL Suite **on UNIX**, the directory selection dialog behaves like described below. The TTCN Suite version **on UNIX** is described in "The TTCN Suite File and Directory Dialog (on UNIX)" on page 31.

- The *Directories* list works in the same way as in the file selection dialog. It is the last directory in the indented list of directories that is chosen when the *OK* button is pressed, unless the text field is specified. This means that it is not enough to select a directory in the list; it has to be double-clicked to become the chosen directory.
- The *Text Field* may be used to enter any directory name, including a relative or absolute path. When `<Return>` or the *OK* button is pressed, it is the directory in this text field that is chosen, if it is specified.
- The *Current* button works in the same way as in the file selection dialog. For more information, see "File Selection Dialog" on page 30.

The TTCN Suite File and Directory Dialog (on UNIX)

Many TTCN Suite dialogs **on UNIX** require the specification of directory paths and file names. A special dialog is used for this purpose.

The example used in the description of the dialog below is taken from a *Log file* dialog, but the principle is the same for all file/directory oriented dialogs.

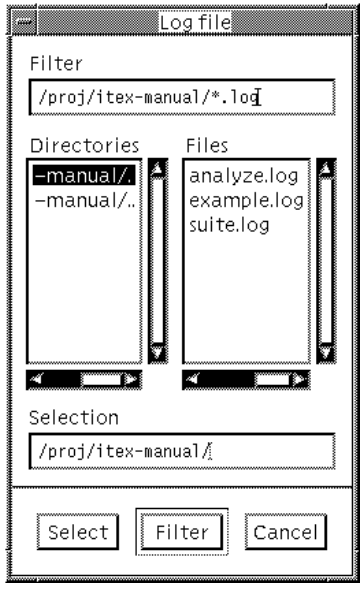


Figure 8: The TTCN Suite file and directory dialog **(on UNIX)**

- The *Filter* field enables you to specify a name pattern which will cause only those file names which match the pattern to be displayed in the *Files* list. Note that the name pattern must be preceded by the complete directory path. This path is set by the TTCN Suite to the currently selected directory (by selecting a directory path in the directories list), but you may edit it directly.
- The *Directories* list shows the directories in the current directory. This list will contain at least two entries, the current directory (a path ending in `/.`) and the parent directory (a path ending in `/..`). To change to another directory in the dialog, select the required directory path and double-click (see also the *Filter* command).
- The *Files* list shows the files that are in the selected directory which match the name pattern. If no files match the pattern or the directory is empty, this is indicated by “[ ]”.

- The *Selection* field displays the currently selected file. This input field also enables you to directly type in a file name.
- The *Filter* button updates the *Files* list depending on the filter pattern (including the file path). Select a directory in the *Directories* list, then choose the *Filter* button to open the directory and display the files in that directory (if any) in the *Files* list.
- The *Select* button terminates the dialog with the selection indicated in the *Selection* input field (in other dialogs this button may have another name, e.g. *Open*, *Export*, etc. but the effect is the same). If there is already a file name chosen before invoking this dialog (i.e. from import dialog when selecting a database name) and no name is specified in the *Selection* input field in this dialog, only the directory path will be modified.

Filename Error Dialogs (UNIX only)

For compatibility reasons, file names on UNIX platforms must not contain colon characters. For an overview on file compatibility issues, see "Windows and UNIX File Compatibility" on page 215 in chapter 2, *The Organizer*.

In the SDL Suite, this restriction is checked whenever the you change a file or directory specification that is stored in the system file (see "System File" on page 189 in chapter 2, *The Organizer* for more information). If they are not followed, an error dialog is shown and you are returned to the dialog where the file was specified.

If you use the TTCN Suite, this restriction affects you only when you choose a name for the system file in the Organizer.

The Busy Dialog

In some special circumstances, a message dialog may be opened, stating that a tool is "busy." This is **not** an indication of a system error. The dialog is opened when a tool is busy performing an operation that you have not finished. A typical example is when you select *Show Organizer* from another tool, and you have not closed a modal dialog in the Organizer, for example the *Print* dialog.

The remedy for a situation like this is to close the message dialog and finish the operation that caused the busy message.

The Timeout Warning

When a tool is started on a heavily loaded computer system, it may fail to start and respond within a certain time limit. When this happens, a timeout warning dialog is opened.

The time limit is specified, in seconds, by the environment variable `STARTTIMEOUT`. The default time limit is 60 seconds. If timeout problems occur, this variable should be adjusted to a higher value to match the typical response times for the computer system where the tool is running. For the changed time limit to take effect, the tools must be restarted.

Keyboard Operations

It is possible to reach and invoke all commands, menu choices and selections by using the keyboard. This section describes the general keyboard operations available.

Menu Traversal

The name of each menu in the menu bar contains an underlined character, as in the menu name *File*. To display, or pull down, the menu, press `<Meta>` **(on UNIX)** or `<Alt>` **(in Windows)** and the corresponding key, for example `<Meta+F>` or `<Alt+F>`.

One character in each menu choice is also underlined. To invoke a menu choice when the menu is displayed, simply press the corresponding key.

The arrow keys can also be used to move between the menu choices and the adjacent pull-down menus.

To invoke a selected menu choice, press `<Return>` or `<Space>`. To cancel the menu traversal and bring down the menu, press `<Esc>`.

Keyboard Accelerators

You can invoke most common menu choices by using an accelerator. An accelerator is a key combination of the form `Ctrl+X`. An accelerator is always case insensitive.

An accelerator always invokes the same kind of command in all SDL Suite tools where it is available. The standard accelerators are:

Accelerator	Functionality or menu command
`Ctrl+F`	Find, *Search* (in the *Tools* menu)
`Ctrl+H`	Help (*On <tool>* in the *Help* menu)
`Ctrl+N`	*New* (in the *File* menu)
`Ctrl+O`	*Open* (in the *File* menu)
`Ctrl+P`	*Print* (in the *File* menu)
`Ctrl+Q`	Quit, *Exit* (in the *File* menu)
`Ctrl+S`	*Save* (in the *File* menu)

Accelerator	Functionality or menu command
`Ctrl+X`	*Cut* (in the *Edit* menu)
`Ctrl+C`	*Copy* (in the *Edit* menu)
`Ctrl+V`	*Paste* (in the *Edit* menu)
`Ctrl+Z`	*Undo* (in the *Edit* menu)
`Ctrl+D`	Scroll one page down
`Ctrl+U`	Scroll one page up

Key Bindings

By using some special keys, you may perform an operation without holding down any modifier key at the same time.

Note:

All keys are not present on all keyboards.

Key	Operation
Arrow keys	Moves the selection in tree structures, list structures and ordinary lists to the closest object in the indicated direction.
`Return (Enter)`	• In a drawing area: the same action as a double-click on the selected object. • In a dialog: the same action as pressing the default button.
`Delete (Remove)`	For texts, clears the character **after** the insertion point. (May be changed with a preference.)
`Backspace`	For texts, clears the character **before** the insertion point.
`F1 / Help`	Opens the help viewer with help on the current window.
`F2`	Raises the popup menu.
`Page Up (PgUp, Prev)`	Moves the visible part of the drawing area a screen upwards.

Key	Operation
`Page Down (PgDn, Next)`	Moves the visible part of the drawing area a screen downwards.
`Home`	Moves the visible part of the drawing area to show the top of the drawing area.
`End`	Moves the visible part of the drawing area to show the bottom of the drawing area.
`Open`	As the menu choice *Open*.
`Find`	As the menu choice *Search*.
`Again`	As the menu choice *Search Again*.
`Undo`	As the menu choice *Undo*.
`Copy`	As the menu choice *Copy*.
`Paste`	As the menu choice *Paste*.
`Cut`	As the menu choice *Cut*.

Key Bindings in the TTCN Suite

More information on keyboard accelerators and key bindings in the TTCN Suite can be found in:

- "Shortcuts" on page 1296 in chapter 30, *Editing TTCN Documents (in Windows)*
- "Key and Button Bindings" on page 1163 in chapter 24, *The TTCN Browser (on UNIX)*
- "Key and Button Bindings" on page 1187 in chapter 25, *The TTCN Table Editor (on UNIX)*
- "Customizing the TTCN Suite (on UNIX)" on page 1243 in chapter 29, *Customizing the TTCN Suite (on UNIX)*

References

[1] Valerie Quercia and Tim O'Reilly:
The Definitive Guides to the X Window System
Volume 3: X Window System User's Guide
OSF/Motif Edition
O'Reilly & Associates, Inc. 1991
ISBN 0-937175-61-7

[2] Open Software Foundation:
OSF/Motif Style Guide Revision 1.2
Prentice Hall, Englewood Cliffs, New Jersey 1992
ISBN 0-13-640616-5

[3] The Open Group:
CDE 2.1/Motif 2.1 – User's Guide
ISBN 1-85912-173-X
`http://www.opengroup.org/publications/catalog/m021.htm`

[4] Microsoft Corporation:
Introducing Microsoft Windows 95
ISBN: 1-55615-860-2

[5] Microsoft Corporation:
Microsoft Windows NT Workstation: Start Here, Basics and Installation

Chapter

2 *The Organizer*

The Organizer is the main tool in SDL Suite and TTCN Suite. When you invoke any SDL Suite or TTCN Suite tool, the Organizer is started and displayed on the screen.

This chapter contains a reference manual for the Organizer; the functionality it provides, its menus, windows and symbols.

Overview

Terminology

The following basic terminology is used throughout this chapter:

- *Document*

  A *document* is a file containing information that can be edited in a particular editor available through SDL Suite and TTCN Suite. Examples of documents in the Organizer are SDL diagrams, MSC diagrams, TTCN documents, and text documents.

- *Generic Document*

  Files containing document types that the Organizer does not normally handle can be included in the Organizer by using a *generic document* symbol and connecting that symbol to the file. The Organizer uses the file extension and the preference variable Organizer*GenericCommand to determine a command to perform when the user tries to edit a generic document.

- *Tau Workspace / Tau Project*

  (Windows only) Files of types .ttw or .ttp that contains a Workspace or a Project handled by the IBM Rational Tau tool. When editing such a document the Organizer will launch the IBM Rational Tau tool.

- *Rhapsody Project*

  (Windows only) Files of types .rpy that contains a Project handled by the IBM Rational Rhapsody tool. When editing such a document the Organizer will launch the IBM Rational Rhapsody tool.

- *Diagram*

  A *diagram* is a graphical document that only can be edited in a graphical editor. Diagrams are either SDL diagrams, MSC diagrams, HMSC diagrams, Object Model (OM) diagrams, or State Chart (SC) diagrams.

- *TTCN Document*

A *TTCN Document* signifies a document of type Test Suite, Modular Test Suite, TTCN Module, or TTCN Package.

- *SDL System*

  An *SDL system* in SDL Suite signifies the topmost SDL diagram with all its sub diagrams, and all used SDL packages with their sub diagrams. In the normal case, the topmost diagram is an SDL system diagram, but the Organizer allows all types of SDL diagrams to be the topmost diagram, as long as the standard hierarchical rules of SDL diagrams are respected.

- *TTCN System*

  A *TTCN system* in TTCN Suite signifies the topmost TTCN document with all its sub TTCN documents.

- *System*

  Not to confuse with an SDL or TTCN system, a *system* in the Organizer refers to a set of documents that, according to your view, are related and thus managed by the Organizer. You decide which documents make up a system and how they are related and grouped together in the Organizer.

  A system can contain documents for analysis, design and testing of one or more SDL and/or TTCN systems. The documents can be textual requirements, analysis and design diagrams, test specifications, source code in different forms, and other types of related documentation.

  In the normal case, one system in the SDL Suite contains one SDL system. But for special purposes it is possible to have more than one SDL system in the system context. Such a purpose could be when working with communicating simulators on UNIX where the source diagrams of each system are required to show and trace the graphical source symbols.

- *Root Document*

  The topmost document in a document structure, or a stand-alone document, or a top-level document in a module. For an SDL system, the topmost diagram in the system, usually an SDL system diagram.

  SDL package diagrams and macro diagrams are special in the sense that they are used by an SDL system diagram. Therefore, they are also placed at the root level and are considered root diagrams.

- *Association*

  An *association* is a link between two documents. Any document can be associated with any other document; a typical example is to associate an MSC diagram with a related SDL diagram. In the document structure, an association symbol is included to the associated document. This symbol is handled like a document, even though it simply represents a document residing somewhere else.

- *Chapter*

  The drawing area of the Organizer is divided into several chapters. You may freely add, remove, and rename chapters, as well as rearrange the order of the chapters. Each chapter has an associated chapter level. It is possible to set the start chapter number.

- *Module*

  A concept specific to the Organizer, used for freely grouping root documents together into a document structure. The user can use modules as a kind of scope unit. Modules have no corresponding concept in SDL, MSC or TTCN terminology. Note, however, that a TTCN Module is not the same concept as a module in the Organizer.

- *Logical Diagram Name*

  The Organizer can identify diagrams in the system by their logical diagram names. The logical diagram name may contain the entire qualified name of the diagram in a specific format. The connection between logical diagram names and physical file names is explicit and under user control.

- *System File*

  A file containing information about the structure and state information of a system as seen by one user. Organizer user settings and viewing options are also stored in this file. The system file is described further in "System File" on page 189.

- *Control Unit File*

  A file that contains a *control unit*, i.e. structure information for a part of a system, common to all users that work with the system. The purpose is to provide workgroup support and a means to put a system under revision control. The user has full control of what parts of a system should be managed as control units. The control unit file is described further in "Control Unit File" on page 200.

- *Link File*

  A file that contains the endpoint and link information, which is managed by the Link Manager. The link file is described further in "The Link File" on page 485.

- *Source Directory*
- *Target Directory*

  The source directory specifies where new documents that you have created, are saved by default, and where to read from when opening and converting documents. **On UNIX,** the source directory is also the directory normally shown when you click *Current* in a standard file selection dialog or a standard directory selection dialog. For more information, see "File Selection Dialog" on page 30 and "Directory Selection Dialog" on page 31 in chapter 1, *User Interface and Basic Operations*.

  The target directory specifies where to put generated files.

  These directories are set in the Organizer, see "Set Directories" on page 71. They are also saved in the system file.

- *Footer File*
- *Header File*

  Footer and header files are used to define how footers and headers should look like when pages are printed. The footer and header file symbols makes it easy to edit the appearance of the footer or header using a text editor. The footer and header file format is described in "Footer and Header Files" on page 344 in chapter 5, *Printing Documents and Diagrams*.

Organizer User Interface

The Organizer consists of two windows: a main window and a log window. For a general description of the user interface, see chapter 1, *User Interface and Basic Operations*.

The main window of the Organizer is also the main window of the tool environment. It manages the current system structure and is responsible for invocation and termination of other tools.

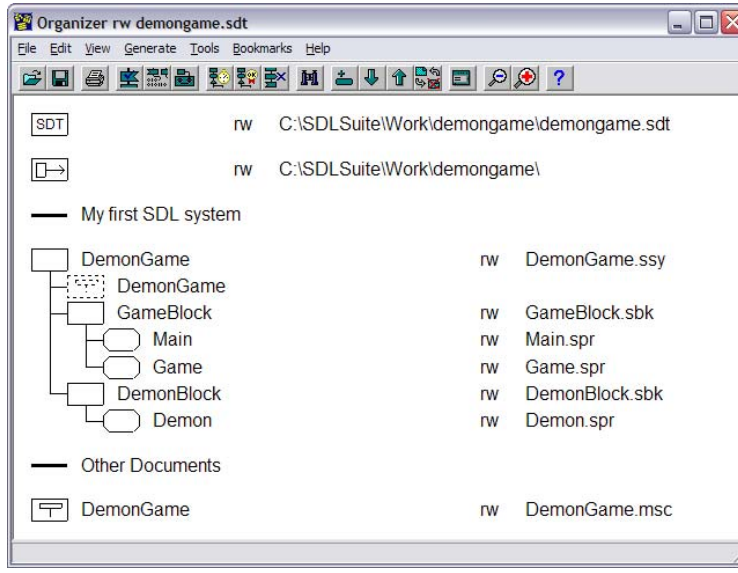


Figure 9: The Organizer main window

The main window exists in one instance only. If more than one Organizer is required, another instance of the SDL Suite or the TTCN Suite must be started. For special purposes, the Organizer can handle more than one TTCN or SDL system simultaneously.

The log window is a text window used for displaying output during analysis and code generation and for displaying general logs. The window is described in "Organizer Log Window" on page 183.

Drawing Area

In the Organizer's drawing area, the files in the system and their structure are presented graphically using icons.

Presentation Modes

The Organizer can present the system information in two different ways graphically:

- As an *indented list*, i.e. a compact, line-oriented list of icons using indentation to suggest the structure (see Figure 10).
- As a *vertical tree*, which is depth- or level-oriented and requires a larger amount of space (see Figure 11).

The user can change the amount of information and the way it is presented in the drawing area with the menu choice *View Options* in the *View* menu. See "View Options" on page 106 for more information.

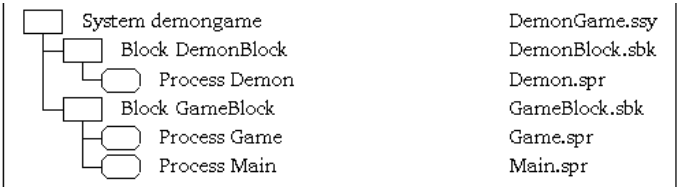


Figure 10: Indented list mode

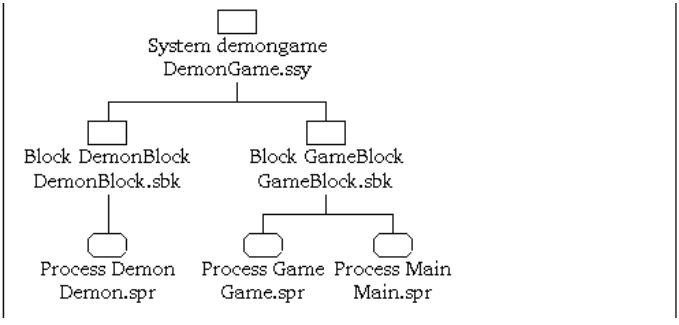


Figure 11: Vertical tree mode

Chapters

The *chapter* concept can be used to define a collection of documents. There are no restrictions on what kind of documents that may be placed in a certain chapter, or on the order of documents in a chapter; this is for the user to decide. Chapters can further be used to structure a print-out of the system included in the Organizer view. More information about this can be found in "Advanced Print Facilities" on page 357 in chapter 5, *Printing Documents and Diagrams*.

A chapter has a name, and possibly a number. A chapter is shown as a thick horizontal line in the drawing area, together with its number and name. A chapter symbol can be placed inside a Module, as well as inside an SDL diagram structure and a TTCN Module structure.

In addition to the line icon, it is possible to have the Organizer show the text "Chapter" ahead of the number and name; see "Icon Names and Types" on page 56.

Chapter Levels

A chapter has an associated chapter level, ranging from 0 to 4, which is reflected in the numbering of the chapter:

- Level 0: a chapter without a number (only a name)
- Level 1: a chapter with a single number: 1, 2, 3, etc.
- Level 2: a chapter with two numbers: 1.1, 1.2, 1.3, etc.
- Level 3: a chapter with three numbers: 1.1.1, 1.1.2, 1.1.3, etc.
- Level 4: a chapter with four numbers: 1.1.1.1, 1.1.1.2, 1.1.1.3, etc.

Numbered chapters are auto-numbered by the Organizer. The initial number for the first chapter in the drawing area can be specified. The initial level of a chapter is specified when the chapter is created, but can later be changed.

Chapter Introduction Text

A chapter can be associated with a text, which can be regarded as an introduction text to the chapter. When the chapter is included in a print-out, this text will be inserted on a separate page corresponding to the position of the chapter symbol.

The text is saved in a text file, which the chapter symbol is connected to.

Default Chapters

There is a default set of unnumbered chapters, known as the *basic Organizer view*, intended for different kinds of documents. This view appears when creating a new system file and contains the following chapters:

- *Chapter Analysis Model*
- *Chapter Used Files*
- *Chapter SDL System Structure*
- *Chapter TTCN Test Specification*
- *Chapter Other Documents*.

The intended use of these default chapters is described below. The default set of chapters can be changed by editing the preference Organizer*Areas.

Chapter Analysis Model

The *Analysis Model* chapter is intended for documents used in a system's analysis, such as Object Model and State Chart diagrams, text requirement documents, MSC, HMSC and SDL requirement diagrams, etc.

Chapter Used Files

The *Used Files* chapter is intended for documents that are not generated but are used, included, or imported by other documents in other chapters. Such documents may be C/C++ header files, etc.

Chapter SDL System Structure

The *SDL System Structure* chapter is intended for the SDL diagram structure that normally is the primary information managed by the Organizer. The diagram structure contains a strictly hierarchical view of one or more root diagrams and their substructures. Most often only one diagram structure is used (a system) with one or more package or macro structures present as separate roots in the system. A characteristic of these diagrams is that they are directly involved in the analysis/code generation process. Multiple system roots are allowed, mainly to support the possibility of running communicating simulators with graphical trace.

Chapter TTCN Test Specification

The *TTCN Test Specification* chapter is intended for TTCN documents used for test specifications.

Chapter Other Documents

The *Other Documents* chapter is intended for documents that do not fit into any of the other chapters. It may contain SDL, MSC, and HMSC diagrams, text documents, etc. that are not directly part of an SDL system and that are not related to the other document structures in the Organizer.

Structure Icons

A few special icons in the Organizer are used for showing how document icons are structured, and for specifying where they are stored in the file system by default.

In addition to the icon, it is possible to have the Organizer show the type of the structure icon textually; see "Icon Names and Types" on page 56.

Different operations are applicable to different types of icons. Double-clicking an icon is described in "Double Clicks" on page 57, and associated popup menus are described in "Pop-Up Menus" on page 170.

System File
The name of the system file. This system file could be managed as a control unit (CM group) to allow revision control of the system file in a multiuser environment.

Link File
The name of the link file.

Control Unit File
This icon designates a control unit whose contents have not yet been resolved.

Source Directory
The name of the source directory.

Target Directory
The name of the target directory.

Header File
The name of the header definition file. Note that it is possible to connect a text file to this symbol, i.e. the symbol can act as a plain text symbol.

Footer File
The name of the footer definition file. Note that it is possible to connect a text file to this symbol, i.e. the symbol can act as a plain text symbol.

Chapter
A chapter in the drawing area. A chapter can have a number and a name and may be connected to a text file.

Module
Another way to define a collection of documents, within a chapter. A module has a name but no associated file name.

More
Indicates that a number of documents are hidden in the document structure (see “Hide” on page 106). The text associated with the icon indicates the number of hidden (additional) documents, e.g. “5 More”.

Bookmark
Quick access to a place in an SDL Suite diagram or document via an SDT reference or to a place on the Web via a URL.

Document Icons

To visualize the different types of documents that are present in the user’s system, a large number of graphical icons are used. A unique icon is provided for every document type that is commonly used in a system. In particular, each SDL diagram type (except packages, systems, macro definitions and operators) already have a given icon (the reference symbol specified in the Z.100 recommendation).

In addition to the icon, it is possible to have the Organizer show the type of the document textually; see “Icon Names and Types” on page 56.

Different operations are applicable to different types of icons. Double clicking an icon is described in “Double Clicks” on page 57, and associated popup menus are described in “Pop-Up Menus” on page 170.

Icon Types

The list below shows the icons that may be part of a system display in the Organizer:

Object Model Diagram icons


Object Model


State Chart

SDL Diagram icons


Package


System


Block


Substructure


Process


Service


Procedure


Macro


System type


Block type


Process type


Service type


Operator


SDL Overview

MSC Diagram icons


MSC


HMSC

Text Document icons

ASN.1 Text

Build Script (see "Build Scripts" on page 2642 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*)

C Header

C Import Specification

C++ Import Specification

Plain Text. Note that connecting a plain text symbol to an external synonym file (*.syn) has a special meaning. See "Update Visibility" on page 102, "Include Expression" on page 2014 in chapter 43, *Using the SDL Editor* and "Supplying Values of External Synonyms" on page 2240 in chapter 50, *Simulating a System*.

Test Script. Used by Organizer>Tools>Simulator Test to execute test cases (in the form of input history scripts *.cui) in the simulator.

SDL PR

Word Text

TTCN Document icons

Modular Test Suite

Module

Package

Test Suite

Flat View

SDL Instance icons

System instance

Block instance

Process instance

Service instance

SDL Dashed icons


Block


Process


Service

Other icons


The icon for an association or a dependency is a dashed version of the icon of the referenced document. The example to the left shows an association referencing an MSC icon.


Generic Document


Tau Workspace / Tau Project


Rhapsody Project


Index icon, representing a generated cross reference file that can be viewed in the Index Viewer.


Page (used for SDL, HMSC, State Chart, and OM diagrams)

Diagram and Document Icon States

The diagram and document icons can have different background colors indicating the state of the information entity associated with the icon. If the icon state is anything else than normal, additional information about the corresponding file is provided in the Status Bar when the icon is selected.


Normal
The normal state of the icon. Information is not modified.
All diagram and document icon types can be normal. This is also the background color used by all non-diagram/document icons.

Invalid
The connected file does not exist or is not a diagram file of the correct type. This state is only possible when connected to a file.
All diagram and document icons can be invalid.
For more information on invalid icons, see "Open" on page 60 and "The Drives Section" on page 192.

Mismatch
The SDL diagram is not referenced in the diagram file where this diagram is referenced in the Organizer, i.e. there is a SDL diagram structure mismatch between the system file and the diagram files.
This state is only possible when connected to a file. Mismatched diagrams that are unconnected are removed from the diagram structure in the Organizer.
SDL diagram icons that are not root diagrams can be mismatched.
The reason for a mismatch is that the reference symbol in the parent diagram states a diagram name different than the kernel heading in the mismatched SDL diagram.
For more information on mismatches, see "Open" on page 60 and "No Save" on page 64.

Dirty
The associated information entity is modified, but not yet saved. This state is only possible when connected to a file.
All diagram and document icons can be dirty.

Ordering

The order of root documents in a chapter is not fixed and can be changed by the user; see "Move Down" on page 182 and "Move Up" on page 182. Moving documents in the Organizer is also used for two other purposes:

- Grouping root documents in a module. Trying to move a root document "over" a non-collapsed module, will result in that the root document becomes a child symbol to the module. A document is also moved out of a module by moving the document up (or down) one or several times.
- Grouping TTCN documents to form a TTCN system. Trying to move a root TTCN document with no children over another (non-collapsed) root TTCN document (possibly with children), will result in that the moved TTCN document will become a child symbol

below the other root TTCN document. A non-root TTCN document is moved out of a TTCN system by moving the TTCN document up (or down) one or several times.

However, the order in which "child" icon types to a "parent" icon appear is fixed according to the following:

1. Page icons
2. Association icons
3. Dependency icons
4. SDL Instance icons
5. SDL Dashed icons
6. Child diagram and document icons

The ordering of page icons is specified in the appropriate editor for the diagram/document type (see "SDL Page Order" on page 1850 in chapter 43, *Using the SDL Editor*, or "Diagrams and Pages" on page 1622 in chapter 39, *Using Diagram Editors*). The ordering of association icons, dependency icons, SDL instance icons and SDL dashed icons is fixed and cannot be changed by the user. The ordering of child diagram and document icons can also be changed.

Icon Names and Types

The names of chapters, modules and documents in the Organizer are always shown to the right of the icon.

It is possible to show the *type* of each icon, i.e. a text of the form "System file", "Chapter", "Module", "Object Model", "Block", etc., directly to the right of the icon (ahead of the name). For SDL diagrams, it is also possible to show the virtuality of the diagram. Both these options are available from the menu choice *View Options* in the *View* menu. See "View Options" on page 106 for more information.

If the document is not opened in an editor, the icon text is presented in a normal, plain type face. If the document is opened in an editor, the text is presented in **bold face**.

Double Clicks

To double click on an icon invokes a default action on the information type that the icon represents.

The menu choices and operations corresponding to the double clicks are:

Type of icon	Menu choice / operation
Diagram and document icons	*Edit* on the selected diagram/document itself
Page icons	*Edit* on the selected page in the associated diagram/document
Association and Dependency icons	*Edit* on the diagram the link points to
SDL Instance icons and SDL Dashed icons	*SDL > Type Viewer* The corresponding symbol becomes selected in the Type Viewer. A double-click on the symbol in the type viewer shows and selects the symbol in the parent SDL diagram.
Source Directory and Target Directory icons	*Set Directories*
System File icon	*Configuration > Group File*
Link File icon	*Link > Link Manager*
Header File and Footer File icons	*Edit* on the connected text file
Chapter icon	*Edit* on the chapter, i.e. brings up the *Edit Chapter* dialog for that symbol
Module icon	*Edit* on the module, i.e. brings up the *Edit* dialog for that symbol
More icon	*Show Sub Symbols*
Bookmark icon	*Edit* on the bookmark, i.e. navigate to the place specified by the associated SDT reference or URL.

Menu Bar

This section describes the menu bar of the Organizer Main window and all the available menu choices.

The menu bar contains the following menus:

- *File Menu*
- *Edit Menu*
- *View Menu*
- *Generate Menu*
- *Tools Menu*
- *Bookmarks Menu*
- *Help Menu*.

Available Menu Choices

The following concepts affect the menu choices that are available in the menu bar.

Long and Short Menus

The user can choose between long and short menus with the menu choice *View Options* in the *View* menu. Menu choices only available in long menu mode are presented with the menu choice name within parenthesis in the textual enumeration of menu choices for a menu in the following sections.

License Dependent Menu Choices

The following menu choices are only available if the corresponding tool is available according to the license configuration:

License	Affected menu choices
Code Generator	*Make*
Simulator	*SDL > Simulator UI*
Explorer	*SDL > Explorer UI*

Configurable Menus

In the SDL Suite, some menu choices may be available through the concept of user-defined menus. For more information, see "Defining Menus in the SDL Suite" on page 18 in chapter 1, *User Interface and Basic Operations*.

File Menu

The *File* menu contains the following menu choices. (Menu choices within parenthesis are not available in short menu mode.)

- *New*
- *Open*
- *Save*
- *Save As*
- *Pack Archive*
- *Unpack Archive*
- *Print > All*
- *Print > Selected*
- *Print > Selected and Colored*
- (*Set Directories*)
- *PC Drives*
- (*Compare System*)
- (*Merge System*)
- (*Import SDL*)
- *Recently used system files*
- *Exit*

New

This menu choice displays a dialog, with the following possibilities:

- Start with a new and empty system, containing the basic Organizer view (see "Chapters" on page 47).
- Start with a standard template system, saved as an archive file (*.tgz) in the SDL Suite installation. The archive files for the standard template systems are and should be saved in directory <install dir>/sdt/include/template/.
- Start with any template system, saved as an archive file (*.tgz) anywhere in the file system.

If you want to start with a template system, you will have to specify the directory to unpack the archive file in, in "Unpack Archive" on page 69 that follows.

If a system file already is open in the Organizer, the behavior is determined by the status of the existing system. If modified information exists, the user first gets the possibility to save it; see "The Save Before Dialog" on page 65.

The new system is then created in memory. *Source Directory* and *Target Directory* are left unchanged, i.e. set to the values they had before the *New* operation.

> **Note:**
>
> The actual value of a directory in the *Set Directories* dialog may change if the directory is set to *System file directory*. Since there is no system file associated with a new system, the SDL Suite and TTCN Suite start-up directory is used until the file is saved.

You have to save the system to create a system file on disk.

The old contents of the drawing area is replaced with the basic Organizer view. If any of the documents in the old system managed by the Organizer were opened in an editor, these editor windows are closed.

Open

This menu choice is usually used to open an existing system file. It can also be used to open a single diagram or document file, as well as to open and resolve a *.scu file.

Opening a System File

If a system file is already open in the Organizer, the behavior is determined by the status of the existing system. If the information is not modified, the *Open* dialog is issued (see below). If modified information exists, the user first gets the possibility to save it; see "The Save Before Dialog" on page 65.

The *Open* dialog is a standard file selection dialog, with the file filter set to `*.sdt`. The *Open* button in the dialog opens the specified system file. The old contents of the drawing area is replaced with the new system. If a system window state file is found, see "System Window State File" on page 199, the window position and size is restored to the position and

state it had when the system file was saved. If any of the documents managed by the Organizer were opened in an editor, the editor windows are closed.

If the system file does not specify the *Source Directory* and/or the *Target Directory* explicitly, these directories are set to the directory where the system file was found.

The following information consistency checks are performed when opening a system:

- That the opened file is a valid system file. The case when an SDT-2 diagram file is opened is discussed above. If a system file created with an earlier version of SDT-3 is opened, you will be warned that the file will be saved in the current format
- All file bindings in the system file are verified. For each file, a check is made to see that the file exists and that the file is of the correct type. The file access permissions are also determined. If something is not correct, it is reported in the Organizer Log.

  If a document is marked Invalid, the user must later correct the file binding. The user could either reconnect the document to a valid file, or perform a disconnection in which case the document disappears.
- The SDL diagram structures in the system file and in the SDL diagram files are also compared, in terms of existing SDL reference symbols. If connected SDL diagrams in the system file are not present in the corresponding SDL diagram files, the diagram icons are marked Mismatch and the status is logged in the Organizer Log window. Unconnected SDL diagrams are removed from the system file and the Organizer's diagram structure. New reference symbols found in the SDL diagram files are added as unconnected diagrams in the system file and the Organizer's diagram structure.

  If an SDL diagram is marked mismatched, the user must later correct the diagram structures. The user could either perform a disconnection in which case the diagram disappears, or change the appropriate diagram in an editor to include the mismatched reference symbol.
- File protection of the system file and the working directory. If either of them is write protected, a warning message appears.

Opening a Diagram or Document File

If you are only interested in examining the contents of a single diagram or document file, you can specify the filename of that diagram or document in the open dialog. (It might be helpful to change the filter in the open dialog first, to be able to view the existing files of the type that you are interested in.)

If you specify the filename of a single diagram or document in the open dialog, then a new system file will be created, only containing the specified diagram or document. The diagram or document will also be loaded and shown in an appropriate editor. Note that no sub(structure) diagram files will be visible in the Organizer view.

*Opening a *.scu File*

Software control unit files *.scu are used to allow several people to work in parallel on the same SDL diagram structure. When you update your *.scu files, for instance by using the Configuration > Update menu choice, the SDL diagram structure might change, if someone else has changed the structure and checked in an updated *.scu file.

You can have a top *.scu file associated with the system file symbol in the Organizer drawing area. This *.scu file will take control of the diagram and document structure in the Organizer, leaving the *.sdt file with only control over the user settings (print, view...) and a little system state information (when was the SDL system analyzed without errors last?)

Opening an *.scu file is the same as creating a new system by attaching a *.scu file to the system file symbol in the Organizer and using Configuration > Update to update the diagram and document structure according to the contents of the *.scu file.

Opening an Archive File

Opening an archive file (*.tgz) invokes the unpack archive file operation. Read more about this in "Unpack Archive" on page 69.

Save

This menu choice saves all modified documents and control unit files known to the Organizer, the link file, and finally the system file used by the Organizer. You can still perform a save even if the Organizer contains a completely new system, or if the system has not been changed

since the last save operation. The menu choice has the text *Save (not needed)* in this situation.

Whenever the system file is saved, the system window state file is saved as well. See "System Window State File" on page 199.

If the system file is modified and needs saving, an asterisk '*' is appended to the name of the system file in the Organizer's title bar.

When the first document that is modified is encountered in the Organizer's view of files, the *Save* dialog below appears. If not *Save All*, *Quit All* or *Cancel* is selected, the dialog will remain on the screen and all modified documents will be handled by the dialog subsequently.

Whether the document is connected to a file or not will affect the layout and behavior of the *Save* dialog. For any unsaved and unconnected documents found, the user must provide a filename to connect to.

If the system has been saved before, the system file is saved (without a dialog) after all diagrams and documents have been saved. If the system never has been saved, the Organizer presents a dialog and proposes a name for the system file; the prefix is the name of the first document in the structure, the extension is `.sdt`.

The fields and buttons in the *Save* dialog are:

- *Save in file*

  If the document is connected, the name of the connected file is shown. If the system file is to be saved for the first time, a proposed name for the system file is shown. The filename can be edited by the user.

  If the document is not connected, the Organizer proposes a filename based on the document name and a file extension corresponding to the document type, making the file name unique in the file system (see "Save" on page 11 in chapter 1, *User Interface and Basic Operations* for more information). The filename can be edited by the user.

  The Organizer will not accept a file name that would overwrite another document or a diagram file that is loaded in an editor.

  The user is prompted to confirm the file name if it is already used by a document included in the document structure.

If the file exists in the file system when the *Save* button is pressed, the user is warned in a message box that the existing file will be overwritten.

If a valid filename is provided, *Save* or *Save All* below also connects the diagram to the supplied file.

- *Save*

  Saves the document file. Then the next file which needs to be saved is shown.

- *No Save*

  Ignores the file without saving it. Then the next file which needs to be saved is shown. If, during the save process, the user saves some SDL diagram files but not others, there is a risk of SDL diagram structure mismatches between the system file and the diagram files. Therefore, a warning dialog with the following alternatives is opened:

  – Clicking *OK* **does not** save the file and continues with the next file.

  – Clicking *Cancel* returns to the original Save dialog, making it possible to save the file.

- *Save All*

  Saves all files (diagram and document files, control unit files, the link file, and the system file) without a confirmation by the user, with the exception of unconnected files, which causes the dialog to appear again.

- *Quit All*

  Quits all files (document files, control unit files, the link file, and the system file) without saving. If, during the save process, the user saves some SDL diagram files but not others, there is a risk of SDL diagram structure mismatches between the system file and the dia-

gram files. Therefore, a warning dialog with the following alternatives is opened:

– Clicking *OK* as a response to an *Exit* operation, completes the exit process without saving any more documents / files.

– Clicking *Cancel* aborts the save process (possibly the exit process) completely.

Save As

This menu choice works as the Save menu choice with the following differences:

- *Save As* is always selectable.
- There is always a Save dialog for the system file.
- The *Save As* menu choice is used to save the system file under a new name. If the system file is saved under the old system file name, the user has to confirm this in a dialog.

The *Save Before* Dialog

Some operations in the Organizer need to save information before the actual operation can be performed. The saving is only performed if modified information exists in the system. In these cases a *Save Before* dialog is opened, which is very similar to a normal *Save* dialog. The dialog title is *Save before* <*command*> and some buttons may behave differently (see Figure 12 on page 71). If not *Save All*, *Quit All* or *Cancel* is selected, the dialog will remain on the screen and all modified files will be handled by the dialog subsequently.

The *Save Before* dialog is opened for the following menu choices:

- *File* menu: *New*, *Open* and *Exit*

  The save process handles modified documents in all chapters. In the case of *Exit*, unsaved diagrams in editors that are not yet in the document structure of the Organizer are also handled.

  If an SDL diagram is modified, has a diagram substructure and some of these SDL diagrams are either opened in an editor or connected to a file, special care must be taken. If the user chooses not to save such an SDL diagram, inconsistencies between the system file and the diagram files may result. The user is warned in a dialog

and may choose to continue, i.e. not to save, or to return to the Save Before dialog. (See "No Save" on page 64 and "Quit All" on page 64.)

- *File* menu: *Compare System* and *Import SDL*

  The save process handles modified documents in all chapters. However, if any modified documents are opened in an editor, only the system file may be saved, not any of the documents.

- *Generate* menu: *Analyze*, *Make* and *SDL Overview*

  The save process handles modified SDL diagrams in all chapters. The buttons *No Save* and *Quit All* are dimmed.

- *Generate* menu: *Convert to PR/MP*

  The save process handles modified SDL diagrams in all chapters. The buttons *No Save* and *Quit All* are dimmed. If the user clicks *Cancel*, the Convert to PR/MP dialog is opened with the GR source diagram toggle dimmed.

Auto Saving

When selecting any of the *Generate* commands *Analyze*, *Make*, *Convert to PR/MP* or *SDL Overview*, the *Save Before* dialog does not appear if the preference AutoSaveBefore is set. However, unconnected and modified documents still require user interaction. If such documents exist, the dialog appears.

Pack Archive

Pack files related to the system loaded in the Organizer into an archive file (*.tgz).

The archive file has the extension *.tgz and is packed using tar and gzip. This means that the archive file can also be unpacked without using the SDL Suite, from the command line, using g(un)zip and tar.

Note:

Packing and unpacking archive files will only work if the SDL Suite can access the external tar and gzip programs, see "Additional required tools and utilities" on page 3 in chapter 1, *Platforms and Products*. The external tools can be pointed out with the preferences "TarCommand" on page 246 in chapter 3, *The Preference Manager* and "GzipCommand" on page 246 in chapter 3, *The Preference Manager*.

The first Pack Archive Dialog

To use the pack operation, have the files you want to create an archive file for in the Organizer and invoke the operation. The first pack archive dialog will appear, where you can do the following things:

- **Pack all diagrams into archive**. This text field is used to specify where the archive file should be saved, and under what name. As default, the archive file is saved in the target directory, with a name based on the SDL system diagram name, and with a *.tgz extension.
- **Diagram files are relative to**. This text field is used to specify a source directory for files put into the archive. As default, the source directory specified in the Organizer is used. Note that files outside this directory, that should be included in the archive file, are put in a separate directory named *external*.
- **Include top directory in archive**. To avoid mixing files from an archive with other files when unpacking, it is possible to create a top directory for all files in the archive when packing. As default, a top directory is created, with the same name as the SDL system diagram that is packed.
- **Save system file for archive in**. With this option, it is possible to create a new system file for the archive, only referencing files that really are packed with file references updated for the archive. Use this option (instead of packing the original system file) to get a portable archive file. As default, an archive system file is created, in the archive top directory, with a name based on the SDL system diagram that is packed.

The second Pack Archive Dialog

When pressing the Next button in the first pack archive dialog, the second pack archive dialog appears, where file types to include can be decided. The following file types are included as default:

- Archive System File
- SDL Package Diagrams
- Other SDL Diagrams
- MSCs
- UML Diagrams
- Chapter Files
- Other Text Files
- Index Files
- Header and Footer Files

The following file types are not included as default:

- Original System File
- Generic Files
- Tau Workspace / Project
- Rhapsody Project
- Simulator Script Files
- *.scu files

The second pack archive dialog also makes it possible to include files from the source or target directories with specific extensions. These extensions should be specified in the **Also include files with the following extensions** text field (a comma separated list of extension names). As default, *.lst files from the source or target directory are included in this way, with the text field text set to *lst*.

The third Pack Archive Dialog

Pressing the Next button in the second pack archive dialog displays the third pack archive dialog, where all files that will be included in the archive are listed. A file in the list is usually presented in the following way:

```
<file system file name> (-> <archive file name> )
```

In the dialog, it is possible to include or exclude individual files.

- To include a file, press the *Add* button and specify the file in the file selection dialog that appears.
- To exclude a file, select it in the list, and press the *Remove* button.

When you are satisfied with the list of files that will be packed, press the *Pack* button to start the pack operation. Information from the pack operation can be found in the Organizer Log.

Unpack Archive

This operation unpacks files in an archive file (*.tgz), and places them in a directory in the file system.

An archive file is created and unpacked with the external tools *tar* and *gzip*, that both must be available for the operation to work. The external tools can be pointed out with the preferences "TarCommand" on page 246 in chapter 3, *The Preference Manager* and "GzipCommand" on page 246 in chapter 3, *The Preference Manager*.

An archive file can be unpacked in one of the following ways:

- By invoking the unpack archive operation in SDL Suite:
  - Select the Organizer > File > Unpack Archive menu choice.
  - Specify a *.tgz file in the Organizer > File > Open dialog.
  - Start SDL Suite with a *.tgz file as a parameter.
- From the command line, using gzip and tar directly. (For more information about this alternative, see the documentation for these tools.

The Unpack Archive Dialog

Whenever SDL Suite is used to unpack, and the unpack directory has not been specified, the Unpack Archive dialog appears. The dialog has three things that should be considered:

- **Unpack archive file**. In this text field, the archive file to unpack should be specified.
- **In directory**. In this text field, the directory to unpack the files in should be specified.
- **Open unpacked system file in Organizer**. If this option is on, the Organizer will after the unpack operation search for a system file (*.sdt) among the unpacked files, and if there is a system file, it will be opened in the Organizer.

Print > All

Similar to *Print > Selected*. The difference is that for *Print All*, the selection is not considered, all diagrams and documents are always printed, if the set of diagrams and documents to print is not further refined in the print dialog.

Print > Selected

Prints all or some of the diagrams in the Organizer. See chapter 5, *Printing Documents and Diagrams*, for more information about the dialog and some examples of how to print. The Organizer selection decides the set of diagrams and documents to print, if the set is not further refined in the print dialog.

Print > Selected and Colored

Similar to *Print > Selected*. The difference is that for *Print Selected and Colored*, only pages (when pages are shown in the Organizer drawing area) or diagrams (when pages are not shown in the Organizer drawing area) with at least one colored symbol are printed. A symbol is colored when the symbol has a border color other than black or a fill color other than white.

Print > Selection File

A file selection dialog appears, where a print selection file should be specified (*.sel). The print selection file is read, and the Organizer print dialog is displayed, updated to reflect the state that the print dialog had when the print selection file was saved from the print dialog. The print selection file is capable of remembering:

- Selected diagrams/documents/pages.
- Print options (scale, paper format...)
- Index Viewer filter options (filter types, uses and diagrams)
- State Matrix Viewer filter options (in Text Editor) (filter processes)

Note that print selection files can also be used by having a print selection symbol in the Organizer. In that case, you can use the print selection file and invoke the print dialog by double-clicking on the print selection symbol.

Set Directories

This menu choice sets the source and target directories. For more information on these directories, see "Source Directory" on page 44 and "Target Directory" on page 44.

This menu choice is only available if the system file can be changed.

If the source directory is changed while a *Save* dialog is active in an editor, the directory where the editor saves the diagram is undefined. An ongoing analysis is not affected by changing the target directory.

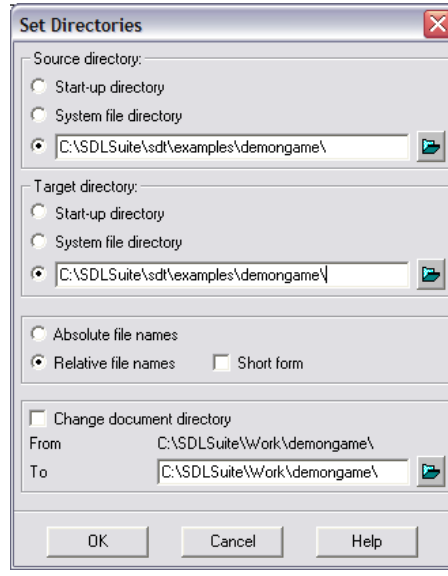


Figure 12: The Set Directories dialog

- *Source Directory*
- *Target Directory*

  Source and target directory settings are saved in the system file. Both these directories can be specified in three ways:

  – SDL Suite start-up directory: Source or target directory is set to the directory where the tool was started from (**in Windows** the directory of the executable file or the specified start directory for a shortcut icon). This setting means that relative file bindings in

the system will be evaluated starting from the start-up directory the next time the system file is loaded into the Organizer.

– *System file directory:* This is the default value for both source and target directory. Source or target directory is set to the directory where the system file resides. For a new system that has not been saved yet, the start-up directory is used until the system is saved. This setting means that relative file bindings in the system will be evaluated starting from the system file directory the next time the system file is loaded into the Organizer. This setting also makes it possible to move a system file and all related diagram and document files to a new directory without having to update the system file, provided that the positions of all diagram and document files relative to the system file are preserved.

– A specific directory. The directory specified in the text field will be used. Note that the target directory can here be specified using a relative file name. If this is done, the source directory is used as a base to dynamically calculate an absolute target directory. This is useful when you want to have a target directory as a sub directory to the source directory and you want to be able to move all your system files in or between file systems without great effort.

- *Absolute file names*

  If this option is set, the Organizer shows and stores document files with full (absolute) path.

- *Relative file names*

  If this option is set (the default), the Organizer stores document files with paths relative to the source directory. There are two variants regarding showing document files in the Organizer drawing area:

  – *Short form* on (the default). The file connection is shown exactly as it is stored in the system file.

  – *Short form* off. The directory part of a file connection is shown in *italics* for a file connection that exactly matches the *Source Directory* path. All other file connections are shown as absolute file names, even if they are not necessarily stored with absolute file paths in the system file.

- *Change document directory*

  This operation is used to change the directory part of one or more file connections in one operation. A symbol that has a file connection including the directory that should be changed has to be selected before invoking the dialog. This operation will be dimmed if there is no selection or if the selected file has no file connection.

  In the *To* text field, the directory which the selected directory should be changed into should be specified. When pressing OK with *Change document directory* on, all file connections matching the *From* directory will be changed to the *To* directory.

PC Drives

This menu choice displays the *drive table* of the currently opened system, i.e. the mapping between drive names in Windows and the beginning of corresponding directory paths on UNIX. For more information, see "Windows and UNIX File Compatibility" on page 215.

This menu choice is only available if the system file can be changed.

The following dialog is opened:

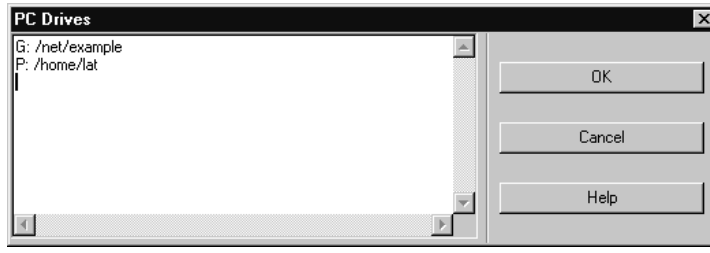


Figure 13: The PC Drives dialog

The text area displays the drive table currently used in the system. The table can be edited directly in the text area. When the system file later is saved, the table is stored in the The Drives Section.

When clicking *OK*, a basic syntax check is performed on the entered drive table. Each line should consist of two items only:

- A Windows path, either a drive letter followed by a colon, e.g. `H:`, or a full path (e.g. `C:\TEMP`). UNC paths can be used (e.g.

`\\<host>\file`). If you include a trailing backslash (optional) you must also include a trailing slash in the corresponding UNIX path.

- A UNIX directory path starting with a slash '/'. If you include a trailing slash (optional) you must also include a trailing backslash in the corresponding Windows path.

Paths containing spaces must be put within double quotes. Note that within double quotes, each backslash must be entered as two backslashes, i.e. a UNC path `\\host\dir name` (containing spaces) must be entered as `"\\\\host\\dir name"`.

If any errors are found, the user is notified and the dialog is not closed. If the table was changed and found to be syntactically correct, the system file is marked as modified.

> **Note:**
> Changes made in the *PC Drives* dialog only take effect when the system file is reloaded.

Compare System

Works as *Merge System*, except that there is no possibility to merge the differences found during the compare operation.

Merge System

This menu choice compares the contents of the Organizer Main window with the contents of a system file (`.sdt` file). The compare operation is performed on a diagram/document level.

This menu choice is only available if the system file can be changed.

The two systems are compared and possible differences are reported to the user, with the option to merge them, by specifying which documents to add and which to remove in the Organizer system.

The information is processed according to the following scheme:

1. The user is asked to exit any editors, if any is found running. This means that modified documents must be saved before the comparison can be started.
2. If the system file is modified, the user is prompted to save it in a The Save Before Dialog.

3. A standard file selection dialog is issued, where the system file to compare the Organizer system with may be selected. Choosing a system file in this dialog starts the compare operation.

4. If there are differences between the options set in the Organizer and the options saved in the system file to compare with, these are reported first, as a text in a separate dialog. The text might look like this:

```
Comparing system in Organizer
with /home/develop/lat/target/DemonGame.sdt.

Options saved in system file differs:

SemanticControl differs: False True
Kernel differs: SCTVALIDATOR SCTADEBCOM
```

   If you want to remove these differences, you have to do it manually by changing different settings in the Organizer. For instance, to remove the SemanticControl difference by updating the Organizer settings, bring up the Analyzer dialog and select *Semantic analysis*. (*SemanticControl* is the word used in the system file, *Semantic analysis* are the words used in the graphical user interface for the same thing.)

5. The *Compare System* dialog is issued, where the diagram and document differences are reported (if the Organizer system and the contents of a system file are found identical, this is reported in a message box and the operation is terminated).

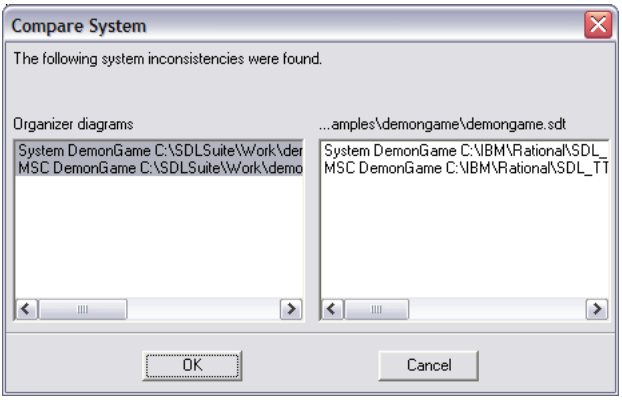


Figure 14: The Compare System dialog

6. The user decides if and how to merge the two views, by selecting or deselecting the items to include or exclude from the resulting system. *OK* updates the system in accordance with the settings.

The *Merge System* Dialog

- The lists are sorted according to the order of appearance of the items in the Organizer Main window and the order of appearance in the system file.
- Each item in a list identifies a chapter, module or document that was found in only one of the two systems (i.e. in the Organizer or in the system file) or were found in both systems, but differ from each other (see "How Systems Are Compared" on page 78 for more information about how structures and documents are compared).

A document is listed with its type, name and the file it is stored on. A module is listed with its type and name. A chapter is listed with its name.

  – A chapter, module or document that was found in one system but missing in the other, is present in one list only and is identified in the other list with a non-selectable '-' (hyphen).

  – Documents that were found in both systems and that are considered *equal* are not listed. See "equal" on page 78 for more information.

  – Documents that were found in both systems and that are considered *almost equal* are presented in both lists. See "almost equal" on page 79 for more information.

- Association and dependency links are not included in the lists, but are preserved as far as possible. See "Associations and Dependencies" on page 79 for more details.
- The items in the two lists that are selected will be included in the resulting (merged) Organizer view.

The items in the *Organizer diagrams* list are the icons that were found in the Organizer's chapters.

By default, all items that originate from the Organizer are selected, meaning that they will be included in the merged view.

Figure 15: A diagram originating from two sources

The MSC DemonGame has been found both in the current Organizer view (where the diagram is connected to a file) and in the system file (where it is connected to another file). The dialog suggests by default to include the diagram originating from the Organizer (the left list) and to exclude the diagram originating from the system file. The user may however change the selection to merge the systems in a different way.

The <*system file*> list indicates the directory and name of the selected system file. The items in this list are the items that were found in the system file.

By default, given two documents that are considered *almost equal*, the dialog will select the document originating from the Organizer, not the document originating from the system file. See example in Figure 15, above.

How Systems Are Compared

Some rules that govern how systems are compared:

- **Relative file names**:

  Relative file names are managed as if they were relative to the *Source Directory* as currently specified in the Organizer.

- Handling of **SDL diagrams** (including packages, excluding macro diagrams):

  SDL diagrams in a tree structure are examined starting from the top and down. When two diagrams are found to differ, their diagram subtree is not examined further, the subtrees being considered as a part of the diagrams that differed.

  This means that if the user chooses to include a diagram that has a subtree, the complete subtree will also be included. Similarly, if the user chooses to exclude a diagram that has a subtree, the complete subtree will be excluded.

  The level of indentation used when listing SDL diagrams in the dialog indicates the structural level at which a diagram is found.

- Handling of **Object Model diagrams**, **State Chart Diagrams, HMSC diagrams, MSC diagrams**, **SDL macro diagrams**, and **SDL overview diagrams**:

  Since these diagrams cannot be part of a subtree, adding or removing any of these does not affect the remaining parts of the structure.

- **Rules for equality of diagrams:**

  Two diagrams/documents with the same type and name can be *equal* or *almost equal*.

  – **equal**

They are considered *equal* if the file names are equal (equal diagrams/documents are not listed in the dialog).

– **almost equal**

They are *almost equal* if the file names are not equal. Two documents that are almost equal are presented on one line in the dialog. For root documents it is possible to select both diagrams on one line in the dialog. For non-root documents it is only possible to select one of the diagrams.

Note:

Compare System and *Merge System* do only compare the structural system information saved in the system file. Compare System and Merge System do **not** compare the document contents, such as page names.

Associations and Dependencies

The Compare System function preserves, as far as possible, the association and dependency links that exist between documents:

1. When an association link is found in a system file, the file names of the two documents are saved.
2. If a document with an association link is selected by the user to be included in the Organizer, a new link is generated if a document with the previously saved file name can be found in the Organizer structure. If such a document cannot be found, the association link is removed from the included document.

Import SDL

Imports an SDL diagram or a number of SDL diagrams in SDT-2 or SDT-3 format, and extracts the diagram structure with the possibility to save it in a system file. The command may, on demand, convert the imported diagrams into SDT-3 format without the need to involve the user for each diagram to convert. It is not possible to import an SDT-3 system file.

This menu choice is only available if the system file can be changed.

To avoid potential name conflicts when saving an imported diagram structure, the menu choice will not perform any action, and causes a message box to be displayed, in the case any files are opened in an editor.

Basically, the involved diagrams and their corresponding files are bound and presented in the Organizer. A number of information entities can be extracted from the diagrams.

If a system file already is open in the Organizer, the behavior is determined by the status of the existing system structure. The user is first asked to exit any editors, if any is found running. This means that modified documents must be saved before the import is started. If a modified system file exists, the user first gets the possibility to save it; see "The Save Before Dialog" on page 65.

The following dialog is then opened:

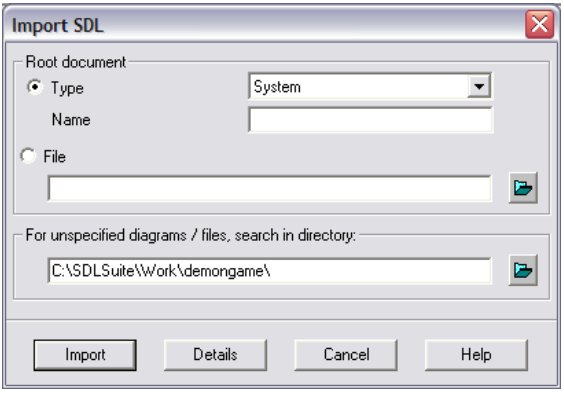


Figure 16: The Import SDL dialog

The root diagram to import and convert can either be named logically with type and name, or physically with a file name. The specified diagram will become a new root diagram, placed last in the Organizer.

- *Type*

  The type of the diagram is selected with this option menu. Used when naming a diagram logically.

- *Name*

  The name of the diagram. Used when naming a diagram logically.

- *File*

  The name of a diagram file. Used when naming a diagram physically.

- *For unspecified diagrams/files, search in directory*

  Specifies that search for diagram files should be done in the specified directory.

- *Import*

  Starts the import operation. Both a root diagram and a search method must be specified in order to start the conversion. If this is not done, an error message is shown. The Organizer log window informs about the progress and result of the conversion.

- *Details*

  Issues this dialog:

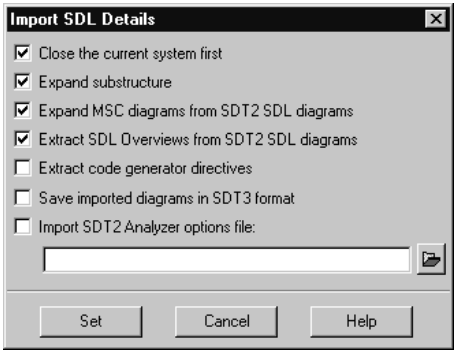


Figure 17: The Import SDL Details dialog

– *Close the current system first*

  If this option is set (the default), the current system in the Organizer will be closed and the SDL diagrams will be imported in a new system. If this option is not set, the SDL diagrams will be imported and added to the current system.

– *Expand Substructure*

  If this option is set (the default), the conversion will be done recursively in the diagram substructure from the specified root diagram. If not set, files corresponding to referenced diagrams are not imported.

– *Extract MSC Diagrams from SDT2 SDL diagrams*

  If this option is set (the default), references to MSC diagrams found in SDL diagrams saved in SDT-2 format will be extracted and placed after the SDL diagram structure, together with a file binding. An association link to the MSC diagram is also inserted in the SDL diagram structure where the MSC diagram was extracted.

– *Extract SDL Overviews from SDT2 SDL diagrams*

  If this option is set (the default), existing overview diagrams stored in SDL diagrams saved in SDT-2 format will be extracted and stored in a separate file. The overview diagram is placed after the SDL diagram structure. An association link to the overview diagram is inserted in the SDL diagram structure where the overview diagram was extracted.

– *Extract code generator directives*

  If this option is set, the diagrams will be searched for a number of directives to be included in the system file. The option is not set as default. The directives #SEPARATE and #WITH are handled

  #SEPARATE: a separator is set in the system structure on each diagram containing the directive (see "Separator symbols" on page 109).

  #WITH: a warning is issued in the Organizer log and a template makefile is generated with the object files found.

  For more information on directives, see "Directives to the Cadvanced/Cbasic SDL to C Compiler" on page 2720 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*.

– *Save imported diagrams in SDT 3.X format*

  If this option is set, parts of the diagrams which are obsolete in SDT-3 are removed from the diagram files to convert them to SDT-3 format. The parts removed are the ones controlled by the options *Extract SDL Overviews from SDT2 SDL diagrams* and *Extract MSC Diagrams from SDT2 SDL diagrams*, described above. If these options are not set, some information may be lost when importing SDT-2 diagrams. In this case, the user will be warned in a dialog when pressing the *Import* button:

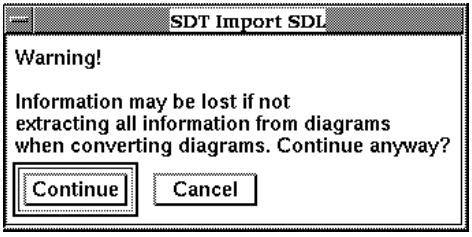


Figure 18: Import diagrams warning

– *Import SDT-2 Analyzer options file*

  If this option is set and a file is specified, the Analyze and Make options found in an SDT-2 Analyzer file (typically `<system>.san`) will be used.

Recently used system files

Just above the *Exit* menu choice, there can be up to four menu choices representing recently opened system files. To open one of them again, select the appropriate menu choice. The information about recently used system files is saved in a file called `.sdtfiles`, in your home directory (see "Environment Variables" on page 45 in chapter 4, *System Setup, in the Installation Guide*).

Exit

This menu choice exits the Organizer.

The exit operation consists of four phases:

1. Handling of modified files managed in the Organizer structure.

   If modified information exists in the current system structure, the user gets the possibility to save it; see "The Save Before Dialog" on page 65. The dialog is opened for the first file (document/system file) that is modified in the Organizer's view of files. The user can then choose how to continue. If the user does not select *Save All*, *Quit All* or *Cancel*, the dialog will remain on the screen and all modified files will be handled by the dialog subsequently.

   The link file and the system file are saved last, if necessary. If the Exit process at a later stage is cancelled, all documents in the editors

are still available, since they are not closed until all modified documents are handled.

2. Confirmation of Exit.

   If no analyze, make, simulation or validation jobs are active, the Organizer exits without user confirmation. If there are such active jobs, or a simulator/explorer UI is executing, the user has the possibility to force an exit of these jobs:

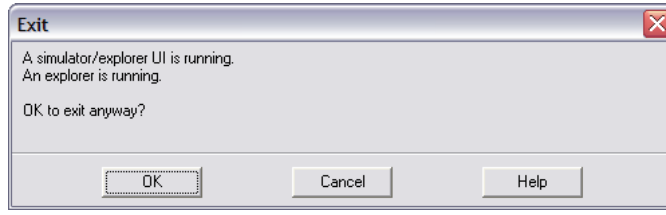


Figure 19: Exit confirmation with active jobs

3. Removing of documents in editors.
4. Shutdown of tools.

   All SDL Suite and TTCN Suite tools are terminated, possibly issuing a *Save before exit* dialog. Finally, the Organizer itself is terminated.

Edit Menu

A general mechanism to edit the document structure(s) in the Organizer does not exist. Some of the menu choices in the *Edit* menu are used for basic operations on root documents and file connections. However, most changes to the document structure are a result of operations made in the diagram editors; see "Reference Symbols" on page 1893 in chapter 43, *Using the SDL Editor*.

The *Edit* menu features the following menu choices (menu choices within parenthesis are not available in short menu mode):

- *Edit*
- *Add New*
- *Add Existing*
- *Remove*

- (*Connect*)
- (*Disconnect*)
- (*Configuration > Group File*)
- (*Configuration > Update*)
- (*Configuration > Full Update*)
- (*Color > Set Default Colors*)
- (*Color > Set Black and White*)
- (*Associate*)
- (*Paste As*)
- (*Go To Source*)
- (*Update Headings*)
- (*Update Visibility*)
- (*Properties*)

Edit

This menu choice edits the selected symbol or document. A document is edited by starting the corresponding editor. A document which is opened in an editor has its name shown in **bold face** in the Organizer.

The menu choice is dimmed if the selected icon is invalid, or if an instance or dashed SDL diagram icon is selected.

The operation performed depends on the type of symbol or document selected, according to the following:

Type of symbol or document	**Operation performed**
System file	The *CM Group* dialog is opened (see "Configuration > Group File" on page 97).
Link file	The Link Manager is opened.
Directory symbol	The *Set Directories* dialog is opened (see "Set Directories" on page 71).

Type of symbol or document	Operation performed
Chapter	The *Edit Chapter* dialog is opened. This dialog contains three choices: • *Edit chapter symbol:* the *Edit* dialog is opened (see below) to allow editing the symbol type, the chapter name, or the connected text file. • *Edit chapter options:* the *Chapter Options* dialog is opened (see "Chapter Options" on page 110). • *Edit first page after chapter:* the first connected SDL diagram or page after the chapter symbol is edited (see below).
Module	The *Edit* dialog is opened (see below) to allow editing the module name.
Connected document or header/footer file	The document/file is opened in an editor. For a diagram, the first page in the diagram is shown, or if the user has specified a page to open first in the editor (see "The Open This Page First Option" on page 2036 in chapter 43, *Using the SDL Editor*), this page is shown instead.
Association or dependency link	The corresponding referenced document is edited as if it was the selected document.
Page symbol	The page is opened in an editor.
Unconnected document or header/footer file	The Edit dialog is opened (see below).

The *Edit* dialog looks like this:

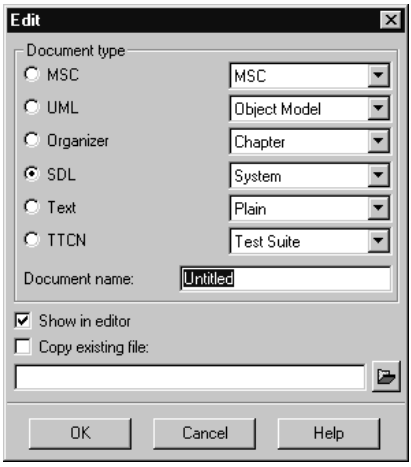


Figure 20: The Edit dialog

- The *Document type* and *Document name* fields can be used to change the type and name of a root document. If a non-root document is selected, these fields are dimmed. If the type is changed, the new document icon will replace the old one. However, the type of a module that contains documents cannot be changed.

  The wanted document type is specified by selecting the correct radio button and selecting the document type in the option menu. It is also possible to select a document in an option menu without selecting the associated radio button first; the radio button will be auto-selected in this case.

- The *Document name* text field contains the current name of the document, chapter or module. The name must conform to the naming rules for the document type; otherwise an error dialog is issued. The name of a module must conform to the naming rules for SDL diagrams, but the name of a chapter may contain any printable characters except quotation marks.

- The *Show in editor* option opens the document in an editor. This option is by default on for document types that can be shown in an editor. Deselecting this option automatically deselects the *Copy existing file* option.

- The *Copy existing file* option copies the specified file and uses it as a starting point/template for the document. If no file is specified, or the file does not exist, no file is copied. Selecting this option automatically selects the *Show in editor* option.

Add New

This menu choice adds a new document to the system. Normally, the added document is placed as a root document below the current selection. If there is no selection, the document is placed as the first root document in the Organizer. Adding a diagram also involves an update of a control unit if there is one that is associated with the diagram substructure affected by the *Add New* command.

There are two exceptions to this:

- If the selection is a module, the new document is instead added at the top level in the module. If the selection is a document in a module, the new document is added in the module at the top level after the selection. To move diagram and documents in and out of modules when they are already in the Organizer, the Move Down and Move Up quick buttons should be used.
- If the selection is a root TTCN document and the user adds a TTCN document, the added document will be placed as a child document below the selected document. If the selection is a child TTCN document, the added document will be placed below the selected document in the same TTCN system. This menu choice is therefore, together with the Move Down and Move Up quick buttons, used to build TTCN systems.

This menu choice is only available if the system file can be changed.

To add SDL diagrams other than root SDL diagrams, the SDL Editor is used (see "Adding a Diagram Reference Symbol" on page 1911 in chapter 43, *Using the SDL Editor*).

The same dialog is opened as for *Edit* on an unconnected document (see Figure 20 on page 88). By default, the *Show in editor* button is on.

If there is a selection, *Document type* and *Document name* in the dialog will be set to the selected symbol's type and name. If there is no selection, the dialog will show the settings from the previous invocation. If

it is the first time this dialog is used, the default type *Module* and the default name “Untitled” will be used.

Multiple root diagrams with the same name are allowed.

If an SDL diagram was selected and an MSC diagram is added, an association link to the MSC diagram is automatically added to the SDL diagram.

Add Existing

This menu choice adds an existing document file to the system. (It is also possible to add several documents by specifying a directory) The existing document is added at the same place as described for the *Add New* menu choice. Adding a diagram also involves an update of a control unit if there is one that is associated with the diagram substructure affected by the *Add Existing* command.

This menu choice is only available if the system file can be changed.

A dialog will be opened, that allows you to specify the file to add, either via a text field or via a standard file selection dialog.

There are also three options in the dialog:

- One to specify if the substructure of the added SDL diagram should be expanded.
- One to specify if the added diagram or document should be shown in an editor.
- One to specify if files existing in sub directories to the specified directory (or the directory where the specified file resides) should be added.

If the standard file selection dialog is used, the file filter is set to reflect the currently selected document type. If a module is selected, the file filter is the same as in the previous invocation. If no document is selected, the file filter is set to `.s??`.

The document type and logical name of the existing document is determined in different ways for different document types. The type and name is determined by:

- Reading the specified file for SDL, MSC, HMSC, OM, SC, and TTCN documents.

- Looking at the file name and extension for text files and generic documents.

> **Note:**
> It is not possible to add a document that does not have a default file extension. For information about default file extensions, see "Save" on page 11 in chapter 1, *User Interface and Basic Operations*.

If an SDL diagram was selected and an MSC or Overview diagram is added, an association link to the MSC or Overview diagram is automatically added to the SDL diagram.

The existing document is by default opened in an editor. This behavior can be changed by the preference Organizer*ShowAddExisting.

Remove

This menu choice removes a selected root document and its document substructure, if any, from the system structure. Modules, chapters and top-level documents in modules can also be removed.

This menu choice is only available if the system file can be changed.

Removing a root document may also involve the update of one control unit file if there is any containing the diagram substructure that has been removed. The menu choice is dimmed if no such document, module or chapter is selected, or if the document is modified.

Removing a chapter symbol does not remove the documents in that chapter; only the chapter symbol itself is removed.

The following dialog is opened:

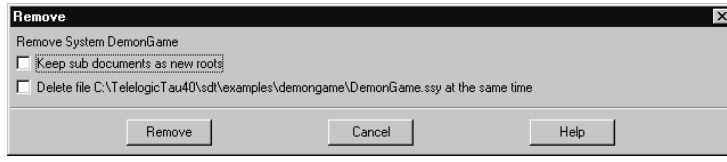


Figure 21: The Remove dialog

- *Keep sub documents as new roots*

If the selected document contains a substructure, this option moves all documents in the substructure to become new root documents (but with their substructures kept intact). It also keeps all bindings to diagrams loaded in an editor. The new root documents are placed directly after the selected root document. If the selected document is a top-level document in a module, the new root documents are placed as top-level documents in the module.

If this option is not set, the document substructure is removed together with the selected document, but documents being edited are still kept as buffers in editors. This is the default setting, except when a module is selected.

The option is dimmed if the document contains no substructure, if no substructure document is connected, or if a chapter symbol is selected.

- *Delete file <file> at the same time*

  This option also deletes the connected file from the file system. It is dimmed if the document is not connected or is opened in an editor, or if a chapter or module is selected.

Connect

This menu choice connects a selected document to a file. It is possible to reconnect an already connected document. The menu choice is hazed if a directory, page, instance diagram, dashed diagram, chapter, or module symbol is selected. The menu choice is also hazed if the loaded system file (*.sdt) or any associated configuration group file (*.scu) file is read-only.

The following dialog appears:

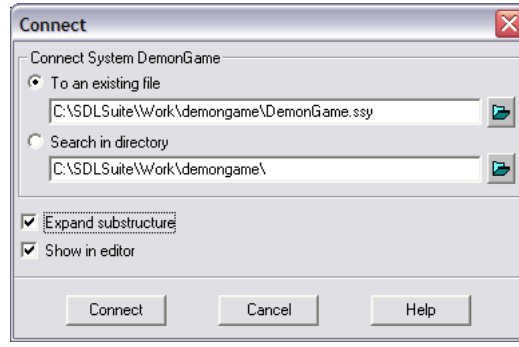


Figure 22: The Connect dialog

- *To an existing file*

  This option connects the document to an existing file. If the document already is connected, the name of the connected file is shown. If the document is unconnected, the field is filled in with the directory component of the file used the last time a document was connected. The filter in the associated file selection dialog corresponds to the file extension for the document type.

  It is possible to use an environment variable to specify the first part of the file path. This can be useful if you want the system to "update its file references" when the environment variable value has changed. For instance, there is an environment variable called "`telelogic`" that points out the top installation directory for SDL Suite when SDL Suite is run. You can use this environment variable to point out one of the SDL diagram files in the installation:

  $telelogic/sdt/examples/demongame/Main.spr

  (Please consider using relative file names first, because that is often a better solution. See "Set Directories" on page 71.)

  When the connection is to be made, the selected file is inspected. For instance, for an SDL diagram, if the file is not an SDL Suite object file, an error message box is issued:

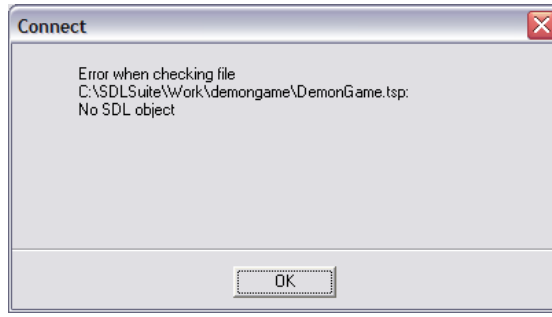


Figure 23: The Connect error message

If the file contains a document of the correct type but with an incorrect name, the symbol in the Organizer is renamed.

If the file contains a document which has an incorrect type or name, the user is warned in a dialog:

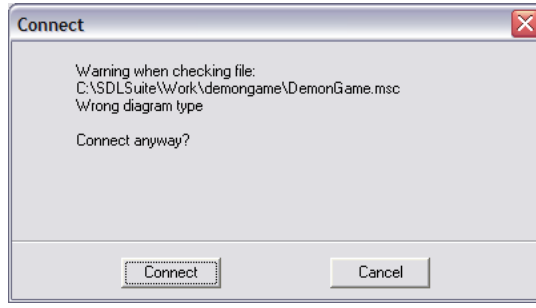


Figure 24: The Connect warning dialog

If the document already is connected and only the directory part of the existing file connection is changed, the following dialog is opened:

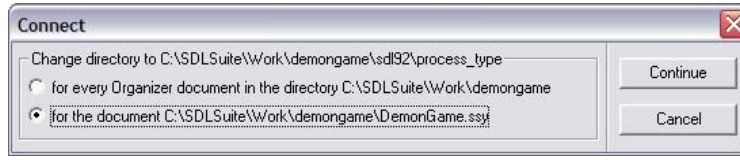


Figure 25: The Connect change directory dialog

- *Search in directory*

  This option searches the specified directory for a document file of a type and name matching the selected document. If such a file is found, the document is connected to the file. If no such file is found, an information box is opened. Pressing *OK* in the dialog returns to the Connect dialog.

  The directory field is filled in with the Source Directory the first time the dialog is used. After that, the directory from the previous usage of the dialog is remembered.

- *Expand substructure*

  This option is only available for SDL diagram symbols. This option recursively expands and connects SDL diagrams to files until no more reference symbols are found. If an SDL diagram has a USE clause, i.e. it references a package diagram, this option also tries to expand the package and put it as a root diagram in the system. The diagram substructure of the package is expanded. Package references are expanded recursively.

  After the expansion is completed, the Organizer display is updated.

- *Show in editor*

  This option opens the document in an editor after the connection has been established.

Error Notification

If an error occurs, the user is informed in a message box and control is returned to the Connect dialog.

Reconnect Connected SDL Diagram

When performing a reconnect to an already connected SDL diagram, the current SDL child diagram references are matched against those found in the connected file. If mismatches are found, icons are marked as such but the structure is kept intact, if possible.

Connect Open Documents

When connecting an unconnected document that is opened and unsaved in an editor, the file name binding is not conveyed to the editor, i.e., the editor binding is lost.

Disconnect

This menu choice disconnects the connected file from the selected diagram. The menu choice is hazed if the selected symbol has no file connection. The menu choice is also hazed if the loaded system file (*.sdt) or any associated configuration group file (*.scu) file is read-only.

The following dialog is opened:

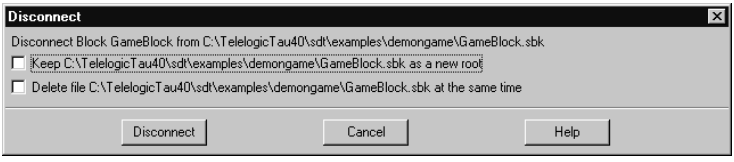


Figure 26: The Disconnect Diagram dialog

- *Keep <file> as a new root*

  This option keeps the connected document in the Organizer structure by making it a new root document. The disconnected document remains in the same place in the Organizer structure.

- *Delete file <file> at the same time*

  This option also deletes the connected file from the file system.

If the document is currently loaded in an editor and is modified, the document reference in the Organizer gets the same status as if a new document is edited, i.e. new and unconnected. The editor binding is then lost.

Configuration > Group File

This command operates on the currently selected diagram, and is used to create or remove a *Configuration Management Group* for the diagram structure. The menu choice is hazed if the loaded system file (*.sdt) or any associated configuration group file (*.scu) file is read-only. When invoked, a dialog appears:

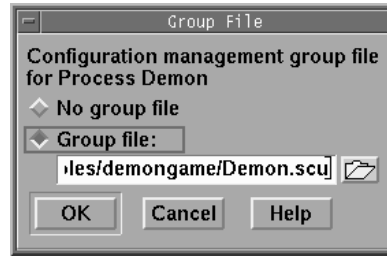


Figure 27: The Group File dialog

- *No group file*

  No group file should be associated with this document; any existing group file for this document group will be removed.

- *Group file*

  Associates a group file with this document, i.e. a *Control Unit* file (*.scu) will be created to hold the structural information about the document and its document substructure. In the text field, the name of the Control Unit file is specified. See "Control Unit File" on page 200 for more information.

  A check is made that the name of the control unit file given by the user is unique within the system. If not, a warning dialogue is issued, making it possible to cancel the operation and to provide a new file name.

  The group file will be shown in the Organizer view like this:

Figure 28: A group file in the Organizer view

The name of the group file is presented directly below the document name, in *italics*. The asterisk '*' indicates that the control unit file is dirty and needs saving. After saving, the asterisk will be removed.

The name of the control unit file is presented directly below the document's file name.

Configuration > Update

A faster version of *Configuration > Full Update*. The *.scu files are only read if they have changed since the last save.

Configuration > Full Update

Updates configuration groups recursively below the selection in the Organizer. Use this menu choice to update the Organizer contents if you have a system with configuration groups loaded in the Organizer and the configuration groups have changed outside the control of the Organizer, for instance by a software configuration management system operation.

Color > Set Default Colors

Invoking this menu choice will make sure that all SDL diagram symbols are colored according to preference values (such as Editor*StateSymbolColor) instead of individual colors set with SDLE > Edit > Symbol Border Color and SDLE > Edit > Symbol Fill Color. Note that this menu choice only operates on the diagrams selected in the Organizer.

Color > Set Black and White

Same as Organizer > Edit > Color > Set Default Colors, but instead of preference values, black and white is used for all symbols.

Associate

This menu choice associates or disassociates a selected document with another document. An association symbol indicates that two document symbols are connected. (A related symbol is the dependency symbol, see "Dependencies" on page 138.)

The menu choice is hazed if the loaded system file (*.sdt) or any associated configuration group file (*.scu) file is read-only.

If an association icon is selected, this menu choice operates on the associated document, not the icon itself. Any document may be associated with any other document, and a document may have more than one associated document.

The following dialog is opened:

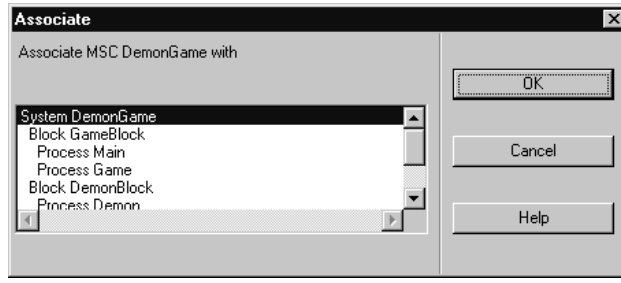


Figure 29: The Associate dialog

- *Associate <document> with*

  The multiple selection list displays the type and name of all documents in the Organizer structure. When the dialog is opened, all documents that the current document is associated with are selected, i.e. the list shows all its associated documents.

  By selecting a new document in the list, an association link to the current document will be created in the selected document's structure. By deselecting a document in the list, the corresponding association link will be removed from the selected document's structure.

Paste As

This menu choice is used to paste copied objects as new diagrams in the Organizer. This menu choice is only available if the system file can be changed. A root diagram is created and opened in an editor. The following transformations are possible via *Paste As* in the Organizer:

- An Object Model class symbol can be pasted as an SDL system diagram.
- An Object Model object symbol can be pasted as an SDL system diagram.
- A text fragment can be pasted as an MSC diagram.

For more information about the Paste As dialog, see "The Paste As Command" on page 448 in chapter 9, *Implinks and Endpoints*.

Go To Source

This menu choice is used to open an editor with a document according to an SDT reference. The SDT reference is specified in a dialog, see Figure 30. If the SDT reference includes information about an object in the document, that object will be selected. SDT references can be obtained by using the menu choice *Show GR Reference* in an editor.

For information about SDT references, see chapter 18, *SDT References*.

The following dialog is opened:

Figure 30: The Go To Source dialog

- *SDT Reference*

  The textual SDT reference is specified in the input field.

- *Go To*

  Shows the symbol in an editor.

An error message appears if the format of the SDT reference is incorrect or if the requested SDT reference cannot be found.

Update Headings

This menu choice checks the headings of SDL, HMSC, OM, and SC diagrams for correctness with respect to what is defined in the Organizer structure.

It operates on the selected diagram and its substructure. If no diagram is selected, all SDL, HMSC, Object Model, and State Chart diagrams in the Organizer are checked. For SDL diagrams, the kernel headings are checked.

Before the headings are checked, a check is made to see if any file is connected to more than one diagram. Such files are reported in the Organizer log, and a warning box is issued to the user. These files may be modified for each appearance and will cause all but the last update to be incorrect.

The heading check is made silently until the first incorrect heading is found. The diagram checked is then shown in the dialog below. If an incorrect qualifier is found, the user is prompted in the dialog whether to update the header or not. The user also has the possibility to silently update all incorrect headings. That is, they are loaded in an editor and are then corrected without confirmation by the user.

After the operation, all updated headings are in an unsaved mode in the editor.

This operation should be done regularly in order to avoid peculiar and hard-to-find analysis error caused by incorrect diagram headings.

In SDL diagrams, qualifiers can be placed in other symbols than the heading in the system, such as qualifying data types in a text symbol. Such qualifiers are not found by the *Update Headings* menu choice.

The following dialog is opened:

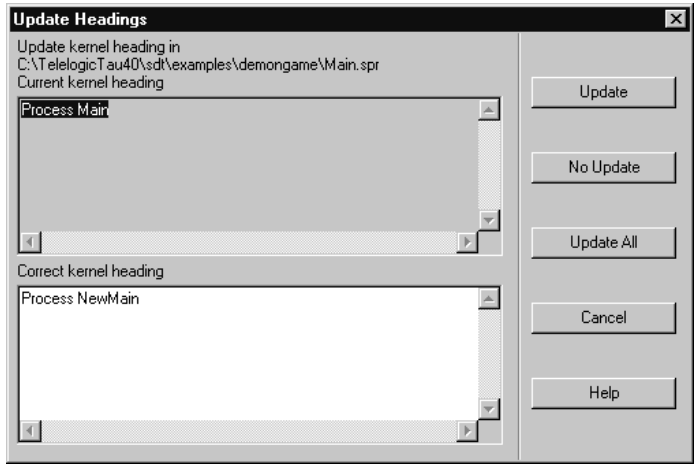


Figure 31: The Update Headings dialog

- *Current kernel heading*

  A read-only text field that shows the contents of the heading as defined in the SDL, HMSC, Object Model, or State Chart diagram, i.e. what is displayed in the editor. (*The Kernel Heading* in SDL diagrams.)

- Correct kernel heading

  A text field with the correct heading according to the structural location of the diagram. This field may be edited by the user.

- *Update*

  Updates the heading in the editor according to the text in the *Correct kernel heading* text field. Continues to search for the next incorrect heading and keeps the dialog open. When the updating is completed, a message appears in the status bar.

- *No Update*

  Does not update the diagram's heading. Continues to search for the next incorrect heading and keeps the dialog open.

- *Update All*

  Updates all headings without a confirmation by the user and closes the dialog. When the updating is completed, a confirmation message is shown in the status bar.

Update Visibility

Update SDL symbol visibility according to include expressions and external synonym values.

The SDL symbol visibility can also be set manually, see "Symbol Visibility > Hide" on page 2014 in chapter 43, *Using the SDL Editor* and "Symbol Visibility > Show" on page 2014 in chapter 43, *Using the SDL Editor*.

For more information about include expressions, see "Include Expression" on page 2014 in chapter 43, *Using the SDL Editor*.

External synonyms are saved in a plain text file with the extension *.syn in the Organizer. To specify that the boolean external synonym variable DEBUG should have the value of true, and the boolean external syn-

onym variable VERSIONTWO should have the value of false, the *.syn file should have the following contents:

```
DEBUG 1
VERSIONTWO 0
```

By using these variable names as include expressions on selected symbols, and by having a reference to the *.syn file in the Organizer, it is possible to hide or show groups of symbols by setting up the correct values in the external synonym file and applying this command. Note that the symbol visibility can not be updated in read-only diagrams.

Properties

Edit bookmark properties for a selected bookmark symbol. *Location* should be set to a valid SDT reference or URL. *Name* is any name that makes it easy to remember the place the bookmark represents. You can get valid SDT references from SDL Suite editors, by selecting a symbol and using <editor>>*Tools*>*Show GR Reference*. The easiest way to create a new bookmark with a valid SDT reference is to use <editor>>*Tools*>*Create Bookmark*.

This menu choice is only available if the system file can be changed.

View Menu

The *View* menu includes the following menu choices (menu choices within parenthesis are not available in short menu mode):

- *Expand*
- *Expand Substructure*
- *Collapse*
- (*Show Sub Symbols*)
- (*Hide*)
- *View Options*
- *Chapter Options*
- (*Set Scale*)
- (*Show High-Level View*)
- (*Show Detailed View*)

Expand

This menu choice expands the symbol structure tree one level down for the selected document. If any symbols one level down are hidden, they

are still hidden after this operation. (Use the menu choice *Show Sub Symbols* to show hidden symbols.)

The menu choice is dimmed if:

- No document is selected
- The selected icon is a leaf (no child icons)
- The selected diagram is not connected
- The selected icon is marked invalid
- The selected icon is already expanded

Expand Substructure

This menu choice expands the symbol structure tree the whole way down for the selected document. This also expands sub symbols that are hidden, but those sub symbols are still hidden after this operation. (Use the menu choice *Show Sub Symbols* to show hidden symbols.)

The menu choice is dimmed if:

- The selected icon is a leaf (no child icons)
- The selected diagram is not connected
- The selected icon is marked invalid
- The selected icon is already expanded

If no document is selected, all icons will be expanded.

Collapse

This menu choice collapses the selected document, i.e. the sub symbols are not shown after this operation. A collapsed document has a small triangle drawn below the icon to indicate that it is collapsed.

The menu choice is dimmed if:

- The selected icon is a leaf (no children icons)
- The selected document is not connected
- The selected icon is marked invalid
- The selected icon is already collapsed

If no document is selected, all icons will be collapsed. A collapsed document does not affect a corresponding document file opened in an editor, i.e. it does not have to be closed or saved.

Show Sub Symbols

This menu choice is used to specify which sub symbols of the selected document that should be shown or hidden. The sub symbols can be documents, instance diagrams, pages, or associations. If a More symbol is selected, the operation applies to the parent document, which becomes selected instead.

Only the sub symbols one level down from the selected document is affected, not the complete symbol substructure. The menu choice is dimmed if there is no selection or the selected symbol has no sub symbols.

Note:

This menu choice does not expand or collapse the document.

The following dialog is opened:

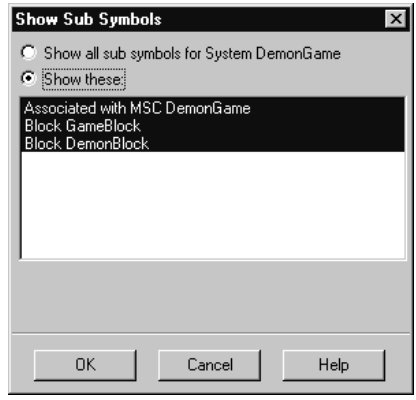


Figure 32: The Show Sub Symbols dialog

- *Show all sub symbols for <document>*

  This option shows all sub symbols for the selected document. It is a shortcut for selecting all sub symbols in the list below.

- *Show these*

  This option shows the sub symbols that are selected in the list, and hides the sub symbols that are unselected. Already shown sub sym-

bols are pre-selected when the dialog is opened. The list is a multiple selection list that toggles the state of the selected item.

Hide

This menu choice hides the selected non-root document and its substructure. The document and its substructure is replaced by a More symbol, which is always placed last of the symbols on that level. If such a symbol already existed in the parent document, the document is hidden under the same More symbol. The symbol's count of hidden documents is updated.

The menu choice is dimmed if there is no selection, or the selected document is a root document.

By double-clicking on the More symbol, the *Show Sub Symbols* dialog is opened.

View Options

This menu choice sets options for controlling the appearance of the Organizer window, as well as options for which icon attributes to show.

The options are set in a modeless dialog, i.e. the Organizer can continue working without waiting for the dialog to be closed. The options are saved in the system file.

The following dialog is opened:

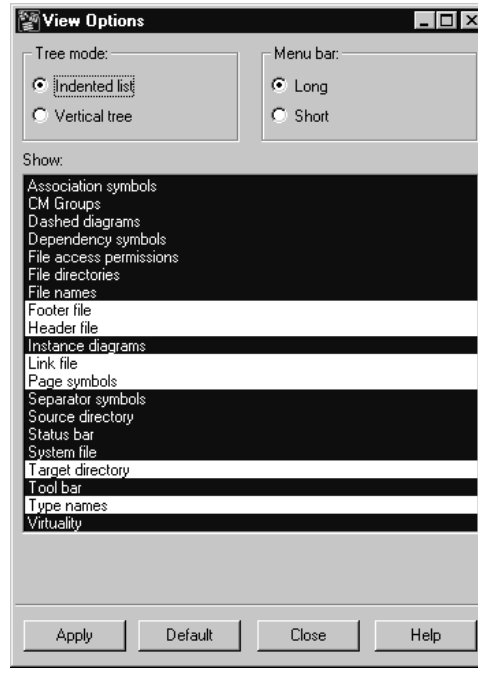


Figure 33: The View Options dialog

The figure above shows the default settings. The settings made in the dialog are preserved as default values the next time the dialog is invoked.

Tree mode

The *Tree mode* section contains options for the two different tree presentation modes available in the chapters (see "Presentation Modes" on page 46).

- *Indented list*
- *Vertical tree*

Menu bar

The *Menu bar* section contains options for which menu choices that are available (see "Long and Short Menus" on page 58).

- *Long*

- *Short*

Show

The *Show* section contains options for which window parts, documents and file attributes to show. The options are available as items in a multiple selection list, which can be selected or deselected. Options already turned on are pre-selected when the dialog is opened.

- *Association symbols*

  Show/hide association symbols (links to associated documents).

- *CM Groups*

  This option governs whether CM Groups should be displayed or not. By default, CM Groups are visible. However, note that they are not shown in vertical tree mode (see "Presentation Modes" on page 46).

- *Dashed diagrams*

  Show/hide dashed diagrams.

- *Dependency symbols*

  Show/hide dependency symbols (links to documents which a document is depending on).

- *File access permissions*

  Add/remove the file access permissions for connected files. An access permission can have one of the following values:

  `rw` Readable and writable
  `r-` Only readable
  `-w` Only writable
  `--` Neither readable nor writable

  In list mode, the permissions are added in front of the file names, if they are shown. In tree mode, the permissions are added on a separate line under each node in the document structure tree, just above the file names if they are shown.

- *File directories*

  Complements the *File names* option below. Add/remove the directory path of the connected files, in the same location as *File names*. If the *File names* option is set, the directory path is added in front of

the file name. The path added is determined by the *Source Directory* option; see "Set Directories" on page 71.

- *File names*

  In list mode: Add/remove a column of connected file names to the documents.

  In tree mode: Add/remove a line of text with the connected filename under each node in the document structure tree. The texts do not overlap; document symbols are separated to make space for the full text.

  If a document is not connected to a file, `[unconnected]` is shown.

- *Footer file*

  Show/hide the Footer File icon.

- *Header file*

  Show/hide the Header File icon.

- *Instance diagrams*

  Show/hide instance diagrams.

- *Link file*

  This option sets the visibility of the Link File icon. The link file icon is hidden by default.

- *Page symbols*

  Show/hide page symbols below SDL, HMSC, OM, and SC diagrams.

- *Separator symbols*

  Show/hide separators between diagrams. Separator symbols show how the generated code will be separated into different files. The lines between diagrams are broken by two vertical or horizontal bars where the separations occur.

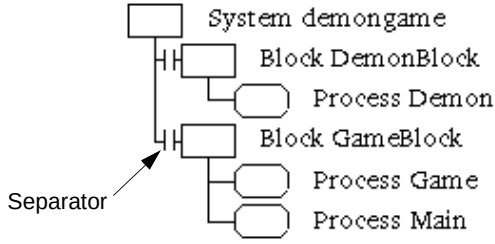


Figure 34: Separators between SDL diagrams

- *Source directory*

  Show/hide the Source Directory.

- *Status bar*

  Show/hide the status bar of the Organizer Main window.

- *System file*

  This option sets the visibility of the System File icon. The system file icon is visible by default.

- *Target directory*

  Show/hide the Target Directory.

- *Tool bar*

  Show/hide the tool bar of the Organizer Main window.

- *Type names*

  Show/hide the type of all icons and documents in textual form, e.g. Object Model, Module, Block, System file, Chapter, etc. The text is placed to the left of the name of the icon.

- *Virtuality*

  Show/hide the virtuality of SDL type diagrams in textual form.

Chapter Options

This menu choice is used to set chapter properties.

The following dialog appears:

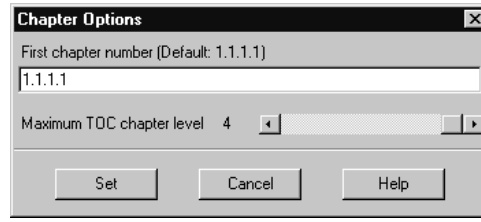


Figure 35: The Chapter Options dialog

The first chapter number is defined in the text field. This number defines the chapter number to use for the first chapter symbol of type Chapter 1, Chapter 1.1, Chapter 1.1.1 or Chapter 1.1.1.1 in the Organizer View area. For example, if the first chapter number is specified as “3.2”, then an initial chapter symbol of type Chapter 1 will get chapter number 3. If the initial chapter symbol instead is of type Chapter 1.1.1.1, it will get a chapter number of 3.2.1.1.

The *maximum TOC chapter level* decides which chapter symbols that will be visible in the table of contents when a print is done from the Organizer. A maximum TOC chapter level of zero will only show chapter symbols of type Chapter, while a chapter level of 4 will show all chapter symbol types in the table of contents.

Set Scale

Sets the scale (20%–800%) used in the Organizer window. The setting is saved in the system file.

Show High-Level View

Sets the current view of all diagrams in the SDL Editor to the high-level view, showing only symbols which are marked as “important”.

Show Detailed View

Sets the current view of all diagrams in the SDL Editor to the detailed view, showing all symbols.

Generate Menu

The *Generate* menu contains the following menu choices (menu choices within parenthesis are not available in short menu mode):

- *Analyze*
- (*Make*)
- *Stop Analyze/Make (UNIX only)*
- *Targeting Expert*
- *SDL Overview*
- *State Overview*
- (*CPP2SDL Options*)
- (*Convert to PR/MP*)
- (*Convert to GR*)
- (*Convert GR to CIF*)
- (*Convert CIF to GR*)
- (*Convert State Chart to SDL*)
- (*Edit Separation*)
- (*Dependencies*)
- (*Merge ASN.1*)

Analyze

This menu choice analyzes the selected SDL or TTCN system. If there is no SDL or TTCN system selected, the Organizer operates on the first SDL system found in the Organizer. The menu choice is dimmed if:

- A job using the Analyzer is already running.
- No SDL diagram is present in the Organizer and no TTCN system is selected.
- The selected SDL diagram or TTCN document is not connected.
- The selected icon is marked invalid.

The Analyze SDL variant of this menu choice is described below, followed by the Analyze TTCN variant.

Analyze SDL

This menu choice starts the Analyzer for one or several related SDL diagrams. If modified information exists in the current system structure, the user should first save it. See "The Save Before Dialog" on page 65 for more information.

Any SDL diagram can be selected for analysis, but in practice at least the parent block diagram will be the source of the analysis. If no SDL diagram is selected, the first SDL diagram found in the Organizer view will be used. The diagram will be analyzed in its context and together with its substructure.

Options for the Analyzer are specified in the modal dialog below. The settings are saved in the system file and persist until the next time the dialog is invoked for the same system.

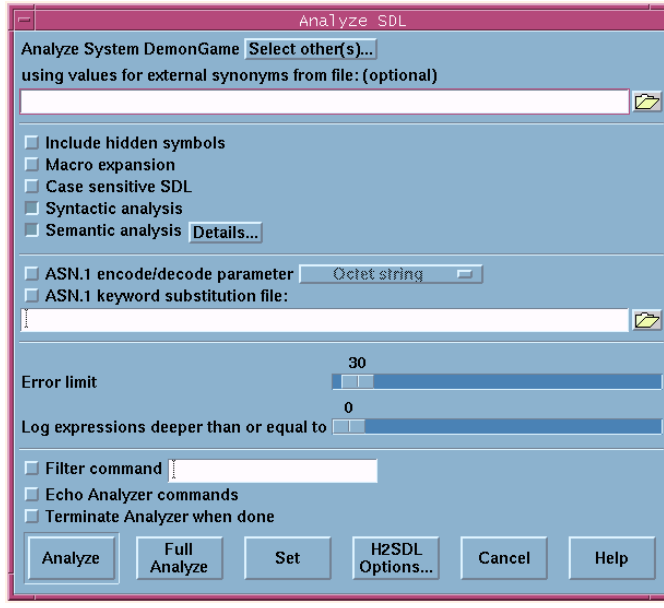


Figure 36: The Analyze dialog

The *Analyze SDL* Settings

Note: Normal versus full analyze (make)

In the *Analyze* and the *Make* dialog, the user can choose between the normal *Analyze*/*Make* and the *Full Analyze*/*Make* buttons. The last used button (normal or full) will be the default when any of these two dialogs is used the next time. The last used button will also determine if normal or full analyze/make will be used when invoking operations via the quick buttons *Analyze*, *Make*, *Simulate* and *Explore*. It is possible to toggle between normal or full analyze/make by pressing `<Ctrl+T>`.

- *Select other(s)*

  When the Analyze dialog is opened, the selection in the Organizer decides the SDL diagrams that will be analyzed. The part that will be considered for analysis is the diagram itself, all diagrams below the diagram and all parent diagrams. A used package is considered to be a parent diagram. All diagrams below a used package will be analyzed.

  This button opens a sub dialog, with the possibility to analyze SDL diagrams other than the one selected in the Organizer. For instance, it is possible to analyze two out of many blocks from the same level in one SDL system. In the sub dialog, just select the diagrams that you want analyzed. Note that you only have to select the top diagram you want analyzed, all sub diagrams will be included in the analysis. The initial label in the Analyze dialog (*Analyze <diagram type> <diagram name>*) will change to reflect the new selections you have made.

- *Using values for external synonyms from file*

  Instead of having a text symbol connected to an external synonym file (*.syn), the external synonym file can be specified here. If there is at least one synonym file in the Organizer view, the one closest to the SDL system will be used as the default value. See "Supplying Values of External Synonyms" on page 2240 in chapter 50, *Simulating a System* for more information.

- *Include hidden symbols*

Decides if hidden SDL symbols should be included in SDL/PR or not. Read more about hidden SDL symbols in *Symbol Visibility > Hide* and *Symbol Visibility > Show*.

If hidden symbols are excluded, then lines to and from hidden symbols will also be excluded. One exception to this is that flow lines going to a symbol that will be excluded, are reconnected to the first following symbol that will be included, if there is one and only one such symbol.

To use include expressions is another way to decide if SDL/GR symbols should be included in SDL/PR or not. Read more about include expressions in *Include Expression*.

- *Macro expansion*

  Run the Macro Expander before the analysis. (See "The Macro Expander" on page 2502 for more information.)

- *Case sensitive SDL*

  Use case sensitive SDL names. If this option is selected, keywords must be all upper or lower case.

- *Syntactic analysis*

  Perform a syntax check.

- *Semantic analysis*

  Perform a semantic check. This option automatically sets the option *Syntactic analysis*.

  Pressing the Details button displays the Semantic Analysis - Details dialog. These options are only used if a semantic analysis is done. They are also not used if a system or package was not the target for the analysis. For more information on these options, see "Perform-

ing Semantic Check" on page 2620 in chapter 55, *Analyzing a System*.

- *Check output semantics*
- *Check unused definitions*
- *Check optional parameters*
- *Check trailing parameters*
- *Check references*
- *Check missing else answers*
- *Check missing answer values*
- *Check parameter mismatch*
- *External types should call GenericFree*
- *Allow implicit type conversion*
- *Include optional fields in make operator*
- *Generate a cross reference file*

  Generate a cross reference file when performing the analysis. In the text field, a file name is proposed with the diagram to be analyzed as prefix and `.xrf` as extension. The file is by default generated in the Target Directory.

- *Generate a complexity measurement file*

  Generate a complexity measurement file when performing the analysis. In the text field, a file name is proposed with the diagram to be analyzed as prefix and `.csv` as extension. The file is by default generated in the Target Directory. See chapter 48, *Complexity Measurements* for more information.

- *Generate an instance information file*

  Generate an instance information file when performing the analysis. In the text field, a file name is proposed with the diagram to be analyzed as prefix and `.ins` as extension. The file is by default generated in the Target Directory.

- *ASN.1 encode/decode parameter*

  This option decides if the ASN.1 encode/decode buffer can be accessed from SDL. If the option is on, it is also possible to specify how the ASN.1 encode/decode buffer should be represented in SDL. For more information, see "SDL Encoding and Decoding Interfaces" on page 2842 in chapter 58, *ASN.1 Encoding and De-coding in the SDL Suite*.

- *ASN.1 keyword substitution file*

  Change keywords in files output by asn1util according to file. If no file is specified, the keywords in the file `asn1util_kwd.txt` in the SDL Suite installation are used. For more information, see "Keywords substitution" on page 705 in chapter 13, *The ASN.1 Utilities*.

- *Error limit*

  The error limit before aborting the analysis (0–1000). A limit of 0 (zero) means that there is no error limit.

- *Log expressions deeper than*

  The performance of the Analyzer depends to a large extent on the depth of the SDL expressions to be resolved. This option is used to emit warnings when the depth of an expression exceeds a certain value (0–100). A limit of 0 (zero) means that no warnings will be issued.

- *Filter command*

  If used, this option allows preprocessing of files before the different analyzer phases. An executable (script), possibly with parameters, should be specified in the text field. The executable will be called before the analyzer processes any file in any phase. The executable will be called with two parameters:

  – The first parameter is the file that is going to be processed.

  – The second parameter identifies the analyzer phase that is going to be executed. Three different phases are distinguished: *import* (before conversion to PR), *macro* (before macro expansion) and *parse* (before syntax analysis).

- *Echo Analyzer commands*

  Print (echo) all Analyzer commands in the Organizer Log as they are executed.

- *Terminate Analyzer when done*

  Terminate the Analyzer process after analysis is done. By default, the Analyzer process is left running in the background after completed analysis.

The options above are forwarded to the Analyzer when the analysis process starts.

The *Analyze SDL* Buttons

- *Analyze*

  Starts the Analyzer in the background and closes the dialog. Status information from the ongoing analysis is shown in the Organizer Log window. Among those diagrams that are considered for analysis according to the top of the dialog (Select other(s)), only the following diagrams are really analyzed:

  – Diagrams that had errors the last time the system was analyzed.

  – Changed diagrams.

  – Diagrams that any diagram mentioned above are depending on.

- *Full Analyze*

  Has the same effect as the *Analyze* button, except that it forces all diagrams that are considered for analysis to be analyzed.

- *Set*

  Saves the Analyzer option settings and closes the dialog, but does not perform an analysis.

Analyze TTCN

This menu choice is used to analyze a TTCN system and/or to generate a Flat View for a TTCN system.

Options for the Analyze process are specified in the modal dialog below. A selectable *Generate Flat View* phase is executed in the analyze process. The settings are saved in the system file and persist until the next time this dialog is invoked for the same TTCN system.

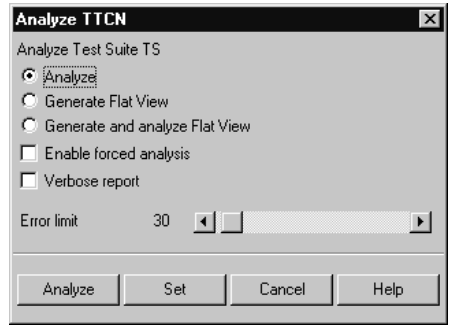


Figure 37: The Analyze TTCN dialog

The *Analyze TTCN* Settings

- *Analyze*

  Selecting this radio button means that the complete TTCN document will be considered for analysis. This radio button is set as default when the dialog is invoked.

- *Generate Flat View*

  Selecting this radio button means that a Flat View will be generated for the selected TTCN system. For a detailed description of this operation, see "Generating a Flat View" on page 1201 in chapter 26, *Analyzing TTCN Documents (on UNIX)*.

- *Generate and analyze Flat View*

  Selecting this radio button means that a Flat View will be generated for the selected TTCN system and it will be considered for analysis.

- *Enable forced analysis*

  This option is described in "Enable Forced Analysis" on page 1193 in chapter 26, *Analyzing TTCN Documents (on UNIX)*.

- *Verbose report*

  This option is described in "Verbosity" on page 1193 in chapter 26, *Analyzing TTCN Documents (on UNIX)*.

- *Error limit*

The error limit before aborting the analysis (0–1000). A limit of 0 (zero) means that there is no error limit.

The *Analyze TTCN* Buttons

- *Analyze*

  Starts the Analyzer and closes the dialog. Status information from the ongoing analysis is shown in the TTCN Suite log. For more information, see "The TTCN Suite Logs" on page 6 in chapter 1, *User Interface and Basic Operations*.

- *Set*

  Saves the Analyzer option settings and closes the dialog, but does not perform an analysis.

For more information about this dialog, see "The TTCN Analyzer" on page 1190 in chapter 26, *Analyzing TTCN Documents (on UNIX)*.

See also chapter 31, *Analyzing TTCN Documents (in Windows)*.

Make

This menu choice makes the selected SDL or TTCN system, or the selected SDL block or process diagram (see "Partitioning" on page 2641 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*). If a Build Script containing commands to the SDL to C Compiler is selected, that file will be used as input to the SDL to C Compiler, without opening the *Make* dialog (see "Build Scripts" on page 2642 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*.

If no document or file of the above mentioned type is selected, the Organizer operates on the first SDL system found in the Organizer. The menu choice is dimmed if:

- A job using the Analyzer is already running
- There is no SDL system diagram in the chapters and no TTCN system is selected
- The selected diagram is not connected
- The selected icon is marked invalid

If modified information exists in the current system structure, the user should first save it; see "The Save Before Dialog" on page 65.

The *SDL Make* variant of this menu choice is described below, followed by the *TTCN Make* variant.

SDL Make

Options for the Make process are specified in the modal dialog below. An analysis phase is executed as part of the Make process. The existing Analyzer options as set in the *Analyze* dialog are used.

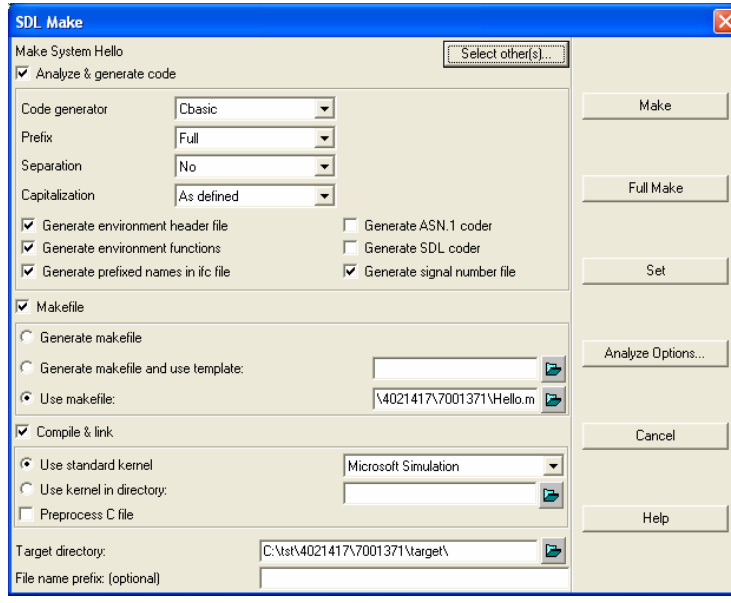


Figure 38: The Make dialog

Code Generation Options

- *Select other(s)*

  When the Make dialog is opened, the selection in the Organizer decides the diagrams that will be used when code is created. This is used to select a partition to build. For a description on partitioning see "Partitioning" on page 2641 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*. This button opens a sub dialog, with the possibility to generate code for diagrams other than those selected in the Organizer. For instance, it is possible to generate code for two out

of many blocks on the same level in one SDL system. In the sub dialog, select the diagrams that you want to include in the code generation process. Note that you only have to select top diagrams, sub diagrams will be automatically included. The initial label in the Make dialog (*Make <diagram type> <diagram name>*) will change to reflect the new selections you have made.

- *Analyze & generate code*

  This option generates code, and performs an analysis if necessary. If not set, the settings below do not affect the Make process.

- *Code generator*

  The code generator is selected with this option menu. The following code generators are supported to date:

  – *Cbasic* (see chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*)
  – *Cadvanced* (see chapter 56, *The Cadvanced/Cbasic SDL to C Compiler* and chapter 61, *The Master Library*)
  – *X*[1] **(UNIX only)**

- *Prefix*

  The type of prefix for variables is selected with this option menu: *Full* (default), *Entity Class*, *No* or *Special*. For more information, see "Prefixes" on page 2736.

- *Separation*

  The type of modularity is selected with this option menu: *No* (default), *Full* or *User Defined*. For more information, see "Selecting File Structure for Generated Code – Directive #SEPARATE" on page 2721.

- *Capitalization*

  The type of capitalization is selected with this option menu: *Lower Case* or *As Defined* (default). For more information, see "Case Sensitivity" on page 2738.

- *Generate environment header file*

1. X is reserved for future extensions in the SDL Suite applied to code generation.

If this option is set, a header file is generated containing the definitions of the SDL system's interface to the environment. For more information, see "System Interface Header File" on page 2777.

- *Generate environment functions*

  If this option is set, environment functions are generated. For more information, see "The Environment Functions" on page 2774.

- *Generate prefixed names in ifc file*

  If this option is set, the names generated in the ifc file will avoid name clashes by being prefixed with `sig_%s_%n`, `lit_%s_%n` etc. For more information, see "Avoiding name clashes" on page 2779.

- *Generate signal number file*

  If this option is set, a file with signal numbers will be generated. For more information, see "Generation of Support Files" on page 2645.

- *Generate ASN.1 coder*

  If this option is set, encoders and decoders from ASN.1 modules will be generated. This option is only available through a special license.

- *Generate SDL coder*

  If this option is set, encoders and decoders from SDL will be generated, see "Type description nodes for SDL types" on page 2791. This option is only available through a special license.

Makefile Options

- *Makefile*

  This option controls makefile creation/usage. If not set, a makefile will not be created/used.

- *Generate makefile*

  Generate a makefile (default). Choose the file by adding a `.m` extension to the separation name of the selected unit and the file is created in the directory specified as target directory. See "Target Directory" on page 71 for more information.

- *Generate makefile and use template*

Generates a makefile and appends the specified, user defined, template at the end of the makefile. Two “hooks” are provided in the generated part of the makefile: USERTARGET and USERLIBRARIES. This enables the user to define his own targets as well as adding properties in the make file. The recommended file name extension for a template file is `.tpm`.

In the template file, USERTARGET is used to add additional object files to the link script in the generated make file, by defining this name as a list of object files.

In the template file, USERLIBRARIES is used to add library modules, for example `-lm` or `-lsocket`, to the link script in the generated make file, by defining this name as a list of libraries.

The template file can also contain the compilation scripts for the object files specified as USERTARGET.

Example 2: Contents of a UNIX Make Template File ––––––––––––––

```
USERTARGET = sctenv$(sctOEXTENSION)
USERLIBRARIES = -lm -lsocket

# Dependencies and actions
sctenv$(sctOEXTENSION): sctenv.c
        $(sctCC) $(sctCPPFLAGS) $(sctCCFLAGS) \
        $(TARGETDIRECTORY) sctenv.c \
        $(sctIFDEF) -o sctenv$(sctOEXTENSION)
```

––

- *Use makefile*

  Specifies an existing makefile to use.

Compile & Link Options

- *Compile & link*

  This option controls if compilation and linking should take place. If set, compilation or linking will be done according to the settings below.

- *Standard kernel*

  Use one of the available standard kernels. The kernel is selected from this option menu. The available kernels depend on the license configuration.

Each kernel is available in versions for different compilers. You should use a kernel corresponding to a compiler you have access to on your system.

On UNIX, the following kernels are available. There are versions for standard ANSI C compilers (e.g. cc) and for the GNU C compiler (gcc).

- *Simulation*
- *gcc-Simulation*
- *RealTimeSimulation*
- *gcc-RealTimeSimulation*
- *PerformanceSimulation*
- *gcc-PerformanceSimulation*
- *Application*
- *gcc-Application*
- *ApplicationDebug*
- *gcc-ApplicationDebug*
- *Validation*
- *gcc-Validation*
- *TTCN-Link*
- *gcc-TTCN-Link*

In Windows, the following kernels are available. There are versions for the Microsoft Visual C++ compiler (cl).

- *Microsoft Simulation*
- *Microsoft RealTimeSimulation*
- *Microsoft PerformanceSimulation*
- *Microsoft Application*
- *Microsoft ApplicationDebug*

Information about the kernels can be found in "Compilation Switches" on page 3119 in chapter 61, *The Master Library* and "Libraries" on page 2770 in chapter 57, *Building an Application*[1]. A list of the available kernels can also be found in the file `sdtsct.knl` (see "File sdtsct.knl" on page 3141 in chapter 61, *The Master Library*).

- *Use kernel in directory*

  Specifies a non-standard kernel to use. The user should specify the directory where the actual kernel is stored.

1. The TTCN-Link kernel is described in the TTCN Suite manuals.

- *Preprocess C file*

  This option informs the SDL to C Compilers if the SDL C Compiler Driver (SCCD) should be invoked. See chapter 60, *SDL C Compiler Driver (SCCD)* for more information.

Miscellaneous Options

- *Target directory*

  This option is by default set to Target Directory. This option determines where the generated files will be put in the file system.

- *File name prefix*

  The names of the generated files will be prefixed by the text put in this field.

Dialog Buttons

> **Note:**
>
> For information about how *Make/Full Make* relates to *Analyze/Full Analyze* and quick buttons, see "Normal versus full analyze (make)" on page 114.

- *Make*

  Starts the code generation in the background and closes the dialog. Status from the ongoing Make process is shown in the Organizer Log window. First the time stamps of the SDL files, e.g *.sbk, *.spr, *.spd, etc, are compared with all their dependent generated c-file counterparts. Only those diagrams that have a newer timestamp are converted from SDL GR to PR, analyzed, and code is generated. After any needed code is generated the make function corresponding to the specified compiler is called to compile and/or link the system. Situations where a make will prove useful are when the following circumstances exist:

  – Diagrams had errors the last time the system was analyzed.

  – Diagrams were changed.

  – Diagrams for which the target files have been removed.

  – Diagrams that any diagram mentioned above are depending on.

- *Full Make*

  Has the same effect as the *Make* button, except that it forces the entire system to be regenerated, even if only certain parts needs to be regenerated. After any needed code is generated the make function corresponding to the specified compiler is called to compile and/or link the system.

- *Set*

  Saves the Make option settings and closes the dialog, but does not perform a make.

- *Analyze Options*

  Opens the *Analyze* dialog to set the analyze options. The *Set* button returns to the *Make* dialog.

TTCN Make

This menu choice is used to make (generate code, compile and link) a TTCN system.

For more information about this operation, see chapter 27, *The TTCN to C Compiler (on UNIX)* or chapter 32, *The TTCN to C Compiler (in Windows)*.

Stop Analyze/Make (UNIX only)

This menu choice stops an ongoing analyze/make operation. The Analyzer tool is also stopped. (The Analyzer tool normally remains resident in memory for the rest of the SDL Suite session once the first analyze/make is performed, and this menu choice is then renamed to *Stop Analyzer*. Using this menu choice is one way to free memory if needed.) Several commands, such as *Analyze*, *Make*, *Convert to PR/MP* and *Convert to GR*, are not available when the Analyzer is processing data. Stopping the Analyzer enables these commands again.

A message with the essence “Analyzer could not be stopped” may be issued as a response to this command; in this case, repeat the menu choice until the message “Analyze/make stopped” is issued in the message area.

Targeting Expert

This menu choice starts the Targeting Expert tool. See chapter 59, *The Targeting Expert* for more information.

SDL Overview

This menu choice generates an SDL overview diagram for the selected SDL diagram as the top diagram. The menu choice is dimmed if:

- No SDL diagram is present in the Organizer.
- The selected SDL diagram is not connected.
- The selected icon is marked invalid.
- The system file is read-only.

If modified information exists in the current system structure, the user should first save it; see "The Save Before Dialog" on page 65.

Any SDL diagram can be selected for generation. If no SDL diagram is selected, the first SDL diagram found in the Organizer view will be used.

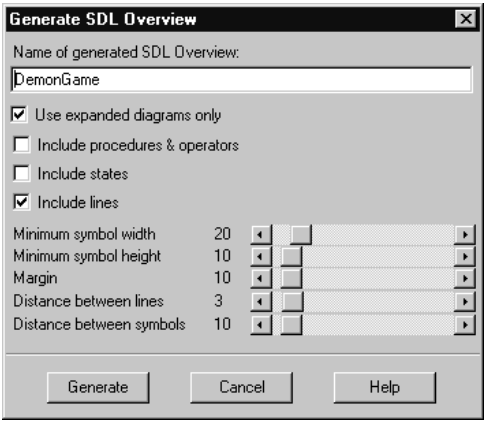


Figure 39: The Generate SDL Overview dialog

- *Name of Generated SDL Overview*

  Specifies the name of the generated Overview diagram. The Organizer proposes a name based on the SDL diagram which the Overview should be generated from.

- *Use Expanded Diagrams Only*

  Controls whether the resulting SDL Overview should comprise all diagrams that are included in the SDL diagram substructure, or only include the diagrams which are expanded (i.e. visible in the Organizer).

- *Include Procedures & Operators*

  Governs if SDL procedure and operator diagrams should be included in the SDL Overview.

- *Include States*

  Controls whether state symbols should be added to the SDL Overview or not.

- *Include Lines*

  Governs whether lines (such as channels and signal routes) should be added to the SDL Overview or not.

- *Minimum Symbol Width*
- *Minimum Symbol Height*

  These slide bars control the minimum size the tool will apply on symbols. (Symbols will be shrunk, if possible, to make the SDL Overview smaller.)

- *Margin*

  This slide bar specifies the distance (in any direction) between symbols at one SDL level and the enclosing frame (the boundaries of the enclosing SDL symbol).

- *Distance between Lines*
- *Distance between Symbols*

  These slide bars define what distance will be inserted between lines that otherwise will overlap each other in the generated SDL Overview diagram.

- *Generate*

  Causes the generation of the overview diagram to start. The user is informed about the progress of the generation in the Organizer Log window. The generated SDL Overview is added as a root document directly after the SDL structure for which the Overview diagram is

generated. An SDL Editor opens and presents the Overview diagram.

State Overview

This menu choice is used to generate a state overview information file from an SDL system or a group of state charts. The state overview can be viewed either as state matrices in the Text Editor or as state charts in the State Chart Editor.

Before selecting this menu choice, make sure that the information source (an SDL system or one or more state charts) is selected directly or indirectly. For instance, you can select a group of state charts by selecting the chapter symbol that contains the state charts.

When you select this menu choice, a dialog is displayed where you can:

- Decide if the information source should be SDL or state charts.
- Change SDL system, if several SDL systems were selected in the Organizer. Use the first *change* button for this.
- Pick out individual state charts, among the state charts that were selected in the Organizer. Use the second *change* button for this.
- Decide name and directory for the generated state overview information file.
- Decide if an Organizer symbol should be created.
- Decide if the Text Editor should show the state overview information as state matrices.
- Decide if the State Chart Editor should show the state overview information as state charts.

CPP2SDL Options

This menu choice is available if a C or C++ Import Specification is selected. For more information, see "The CPP2SDL Tool" on page 761 in chapter 14, *The CPP2SDL Tool*.

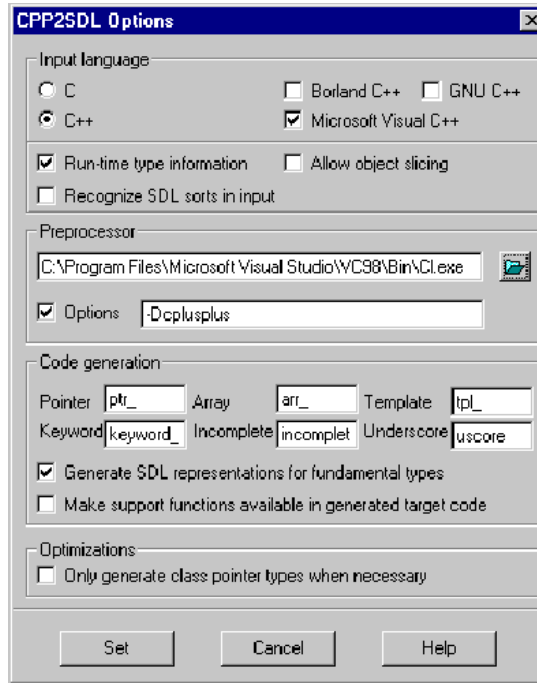


Figure 40

Convert to PR/MP

This menu choice converts the selected diagram/document to textual form. SDL/GR is converted to SDL/PR and TTCN-GR is converted to TTCN-MP. If no diagram/document is selected, the Organizer operates on the first SDL system found in the Organizer. The menu choice is dimmed if a job using the Analyzer is already running.

The *Convert to PR (SDL)* variant of the menu choice is described below, followed by the *Convert to MP (TTCN)* variant.

Convert to PR (SDL)

The SDL variant of the *Convert to PR/MP* menu choice generates a formatted (pretty printed) SDL/PR file. Input is either an SDL/PR file or an SDL/GR diagram structure.

If modified information exists in the current system structure, the user should first save it; see "The Save Before Dialog" on page 65.

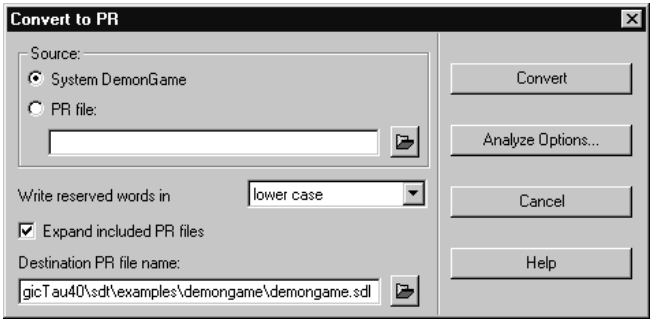


Figure 41: The Convert to PR dialog

- *Source:*
  <type> <name>

  Converts an SDL/GR diagram structure to PR. The top of the diagram structure appears in the name of the button.

  If an SDL diagram is selected, that diagram will be converted. Otherwise, the first SDL diagram structure found in the Organizer will be converted. If no SDL diagrams are found, this option is dimmed, in which case the name of the button is "<No SDL/GR diagram selected>". This will also be the case if the SDL diagram is not connected, or if its icon is marked invalid.

  If the SDL diagram to be converted has an expanded diagram substructure visible in the Organizer, this substructure is also converted. All PR code is put in the destination PR file.

- *PR file*

  Generates a pretty printed PR file from an existing PR file, which is specified in the input field. By default, this file is read from the Source Directory.

- *Write reserved words in*

  Specifies whether reserved words in SDL are to be written in lower case or upper case.

- *Expand included PR files*

  Expands SDL/PR include files found in `/*#INCLUDE...*/` comments.

- *Destination PR file name*

  Specifies the pretty printed PR file to generate. The default name uses the selected diagram name as prefix and `.sdl` as extension. By default, this file is stored in the Target Directory.

  If no file name is provided, the user is warned and no conversion is performed.

  An Overwrite confirmation dialog is issued if the user changes the suggested file name and specifies a file that already exists.

- *Convert*

  Generates the pretty printed PR file and closes the dialog. Status from the ongoing conversion process is shown in the Organizer Log window.

- *Analyze Options*

  Opens the *Analyze* dialog to set the analyze options. The *Set* button returns to the *Convert to PR/MP* dialog.

Convert to MP (TTCN)

This TTCN variant of the *Convert to PR/MP* menu choice converts a TTCN-GR document to a TTCN-MP text file.

For more information about *Convert to MP*, see "Exporting a TTCN Document to TTCN-MP" on page 1152 in chapter 24, *The TTCN Browser (on UNIX)* or "Converting to TTCN-MP" on page 1293 in chapter 30, *Editing TTCN Documents (in Windows)*

Convert to GR

This menu choice converts a textual file to one or more graphical diagrams. A TTCN-MP file is converted to TTCN-GR diagrams, and an SDL/PR file is converted to SDL/GR diagrams. The menu choice is dimmed if a job using the Analyzer is already running.

The *Convert to GR (SDL)* variant is described below, followed by the *Convert to GR (TTCN)* variant.

Convert to GR (SDL)

The *Convert to GR* dialog is in SDL mode when the radio button *Convert* SDL/PR to SDL/GR is on (see Figure 42).

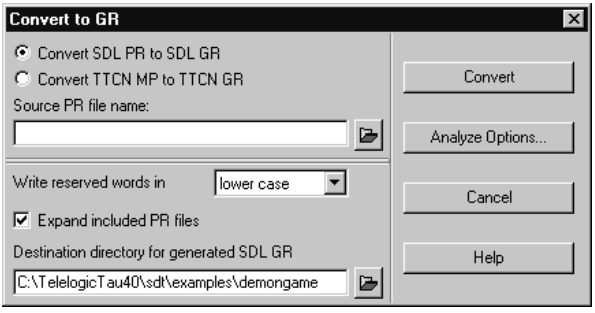


Figure 42: The Convert to GR dialog in SDL mode

- *Source PR file name*

  Specifies the PR file to convert.

- *Write reserved words in*

  Specifies whether reserved words in SDL are to be written in lower case or upper case.

- *Expand included PR files*

  Expands SDL/PR include files found in `/*#INCLUDE...*/` comments.

- *Destination directory for generated SDL/GR*

  Specifies the directory where to put the generated SDL diagrams. By default, the Source Directory will be used (see "Set Directories" on page 71).

- *Convert*

  Generates the SDL/GR diagrams. The SDL/GR files are named in the same way as when saving unconnected documents in the Orga-

nizer, i.e. files are named so that no files are overwritten. Status from the ongoing conversion process is shown in the Organizer Log window.

- *Analyze Options*

  Opens the *Analyze* dialog to set the analyze options. The *Set* button returns to the *Convert to GR* dialog.

The *Convert to GR* command can be used to import SDL systems created using Object Geode. However, Object Geode allowed some extended SDL syntax that is not accepted by the SDL Suite Analyzer.

Note:

In Object Geode, strings can be surrounded by double quotes. This is not accepted by the SDL Suite, only single quotes are allowed. E.g. the statement

```
Writeln("Take your card")
```

must be changed to

```
Writeln('Take your card')
```

In Object Geode it is allowed to have an Operator call with an empty string. E.g. the statement

```
j := tstOperator()
```

must be changed to

```
j := tstOperator
```

to be accepted by the SDL Suite.

In Object Geode you can split the system into many CIF files, where a special #REF directive is used in a comment for the reference symbol. This construct can cause a loop forever during import. A possible workaround is to change the following constructs:

```
PROCESS P1 REFERENCED
COMMENT '#REF <some_path>\<some_file>.cif.pr';
```

into:

```
PROCESS P1 REFERENCED;
```

Convert to GR (TTCN)

The *Convert to GR* dialog is in TTCN mode when the radio button *Convert TTCN-MP to TTCN-GR* is on (see Figure 42).

For more information about the Convert to GR dialog in TTCN mode, see "Importing a TTCN-MP Document" on page 1157 in chapter 24, *The TTCN Browser (on UNIX)* or "Converting to TTCN-MP" on page 1293 in chapter 30, *Editing TTCN Documents (in Windows)*.

Convert GR to CIF

This menu choice converts SDL/GR diagrams to the Common Interchange Format (CIF).

On UNIX, the *Convert GR to CIF* dialog is opened. For more information, see "Convert GR to CIF Dialog (UNIX only)" on page 904 in chapter 16, *CIF Converter Tools*.

In Windows, the SDT2CIF converter tool is started. For more information, see "Graphical User Interface (Windows only)" on page 906 in chapter 16, *CIF Converter Tools*.

Convert GR to Tau/Developer CIF

This menu choice generates a CIF file for an SDL system that should be imported into IBM Rational Tau. The command is a shortcut for the *Convert GR to CIF* command where it is only needed to specify the system file and the resulting CIF file. The functionality is the same as running *Convert GR to CIF* with the following options:

- "Generate one CIF file" on
- "Include graphical SDT references" off
- "Include CIF comments" on
- The input file must be a system file (*.sdt)
- The output file will be the same as the input with extension .cif if not explicitly specified

Convert CIF to GR

This menu choice converts diagrams in Common Interchange Format (CIF) to SDL/GR diagrams.

On UNIX, the *Convert CIF to GR* dialog is opened. For more information, see "Convert CIF to GR Dialog (UNIX only)" on page 889 in chapter 16, *CIF Converter Tools*.

In Windows, the CIF2SDT converter tool is started. For more information, see "Graphical User Interface (Windows only)" on page 890 in chapter 16, *CIF Converter Tools*.

Convert State Chart to SDL

This menu choice transforms the selected State Chart to an SDL process diagram. For more information, see "Converting State Charts to SDL" on page 1702 in chapter 39, *Using Diagram Editors*.

Edit Separation

This menu choice inserts or edits a separation on the selected SDL diagram. It is dimmed if the selected diagram type is not one of system, system type, block, block type, process, process type, procedure, and package. This menu choice is only available if the system file can be changed.

Diagram separation symbols are used during code generation and controls both the splitting of the target into separate modules and the naming of these modules. For more information, see "Selecting File Structure for Generated Code – Directive #SEPARATE" on page 2721 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*. Separations can be shown in the diagram structure of the Organizer; see "Separator symbols" on page 109. Information about separations are stored in the system file.

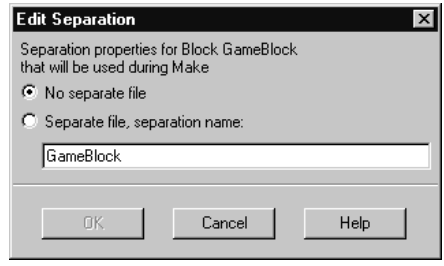


Figure 43: The Edit Separation dialog

- *No separate file*

  No separation is to be used on this diagram; any existing separation is cleared. This option is disabled on system and package diagrams.

- *Separate file*

  Inserts a separation on this diagram, i.e. the diagram and its substructure is generated separately. In the text field, the name of the separation is specified, i.e. the prefix of the files generated for this separation.

Dependencies

This menu choice introduces or removes dependencies between a selected document and other documents. A dependency symbol below a document indicates that the document is depending on another document. For instance, if an SDL system is depending on an ASN.1 document, then the SDL system must be re-analyzed each time the ASN.1 document is updated. A related symbol is the association symbol, see "Associate" on page 98.

This menu choice is only available if the system file can be changed.

> **Note:**
>
> Dependency links between SDL systems and C header files or ASN.1 documents are **not** required, but serve mainly as comments. The SDL Analyzer will re-analyze the SDL system automatically for such dependencies.

If a dependency icon is selected, this menu choice operates on the document the dependency icon is referring to, not the icon itself. Any document may depend on any other document, and a document may have be depending on more than one document.

The following dialog is opened:

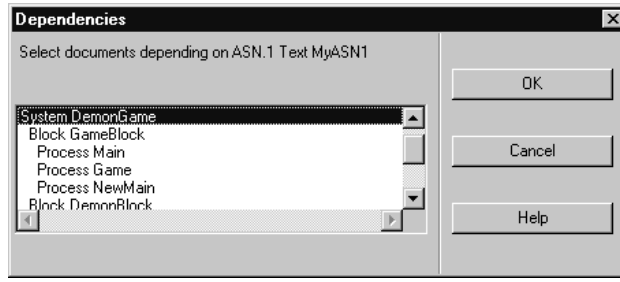


Figure 44: The Dependencies dialog

- *Select documents depending on <document>*

  The multiple selection list displays the type and name of all documents in the Organizer structure. When the dialog is opened, all documents that the current document is depending on are selected, i.e. the list shows all dependencies for the document.

  By selecting a new document in the list, a dependency for the current document will be created. By deselecting a document in the list, the corresponding dependency link will be removed from the selected document's structure.

Merge ASN.1

This menu choice controls whether the ASN.1 files in an Organizer module should be merged (joined) into a single SDL package or not. For more information, see "Using the ASN.1 Utilities" on page 701 in chapter 13, *The ASN.1 Utilities*.

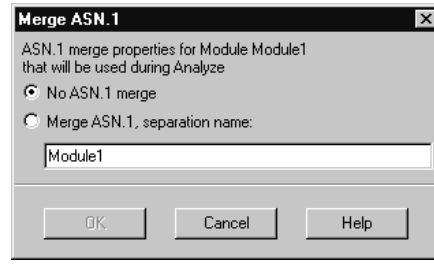


Figure 45: The Merge ASN.1 dialog

- *No ASN.1 merge*

  The ASN.1 files in this Organizer module should not be merged; a separate SDL package is generated for each of the ASN.1 files.

- *Merge ASN.1, separate name*

  The ASN.1 files in this Organizer module should be merged; one SDL package will be generated, containing all ASN.1 modules from the ASN.1 files. The name of the SDL package will be the same as the name of the Organizer module, and the separate name will be used for the code generation in a similar way as for ordinary ASN.1 files.

Tools Menu

The *Tools* menu contains the following menu choices:

- *Organizer Log*
- *Link > Create Endpoint*
- *Link > Traverse*
- *Link > Link Manager*
- *Link > Clear Endpoint*
- *Search*
- *Spelling > Comments*
- *Spelling > All Text*
- *Change Bars*
- *Compare > SDL Diagrams*
- *Compare > MSC Diagrams*
- *Compare > HMSC Diagrams*
- *Merge > SDL Diagrams*
- *Merge > MSC Diagrams*
- *Merge > HMSC Diagrams*
- *Split*
- *Join*
- *Compare State Machines*
- *Simulator Test > New Simulator*
- *Simulator Test > Existing Simulator*
- *Editors > Deployment Editor*
- *Editors > HMSC Editor*
- *Editors > MSC Editor*
- *Editors > OM Editor*
- *Editors > SDL Editor*
- *Editors > State Chart Editor*
- *Editors > Text Editor*
- *Editors > TTCN Browser*
- *SDL > Type Viewer*
- *SDL > Coverage Viewer*
- *SDL > Index Viewer*
- *SDL > Simulator UI*
- *SDL > Explorer UI*
- *SDL > Target Tester UI*
- *TTCN > Find Table*
- *TTCN > Access*
- *TTCN > Simulator UI*
- *Preference Manager*

Organizer Log

This menu choice raises the Organizer Log window. The Organizer Log window can be raised automatically when the user performs an analysis or other forms of generation. There is only one Organizer Log window.

The Organizer Log window is described in "Organizer Log Window" on page 183.

Link > Create Endpoint

This menu choice creates an endpoint for the selected document. A document with an endpoint is recognized by a small triangle in the upper left corner of the document symbol. See "Link > Create Endpoint" on page 444 in chapter 9, *Implinks and Endpoints*.

This menu choice is only available if the system file can be changed.

Link > Traverse

This menu choice traverses a link for the selected document. A document with at least one link is recognized by a small black triangle in the upper left corner of the document symbol. See "Link > Traverse" on page 444 in chapter 9, *Implinks and Endpoints*.

Link > Link Manager

This menu choice opens the Link Manager's main window. For more information about the Link Manager, see "The Link Manager" on page 462 in chapter 9, *Implinks and Endpoints*.

Link > Clear Endpoint

This menu choice removes an existing endpoint for the selected document. See "Link > Clear Endpoint" on page 447 in chapter 9, *Implinks and Endpoints*.

This menu choice is only available if the system file can be changed.

Search

This menu choice searches for text in SDL, MSC, HMSC, Object Model, and State Chart diagrams. Textual documents (C Header, Text ASN.1, and Text Plain) are also searched if the Text Editor is used. (The Text Editor is used if the preference variable SDT*TextEditor is set to

"SDT.") It is not possible to search in TTCN, Word, Tau/Rhapsody or Generic documents with this menu choice. To search in TTCN documents, use *TTCN > Find Table*.

The document scope of the search depends on the selection:

- If a module is selected, all searchable documents in that module will be searched.
- If a chapter is selected, all searchable documents in that chapter will be searched.
- If a root document is selected, all searchable documents in that document's substructure will be searched.
- If there is no selection, all searchable documents in the Organizer will be searched.

The menu choice is dimmed if the selected document is not a root document, not connected, or marked invalid.

The search will only take place in diagrams that are connected and do not have an invalid status. The search will start in the selected diagram and continue in top-down order for the rest of the diagrams (the order is left-right in a tree view).

The search function will go through the list of diagrams and stop each time the search criteria, as set in the dialog below, matches. If a search/replace string or any option is changed when the search is stopped (a match is found or the user pressed *Abort*), these values become the basis when the search is continued.

The search process will open an editor window, if necessary, and select the matched search text.

The searching is based on ASCII character matching. All text fragments in symbols are searched, with a few restrictions (see below).

When all diagrams have been searched, a beep is issued and the message *Search completed* appears.

Dialog Fields and Options

The *Search* dialog contains the following fields and options.

- *Search for*

The text to search for. To the left of the text field, there is an option menu containing old search strings. To search again for a string that has already been searched for, select the search text in the option menu.

- *Replace with*

  The text which is to replace the text searched for. Does not have to be specified.

- *Search in*

  Two option menus where it is possible to restrict the search to a selected diagram or symbol type. For instance, you can find all SDL input symbols by not specifying any search text, and specifying "SDL" as a diagram type and "Input" as a symbol type.

- *Consider case*

  If this option is set, search is case sensitive.

- *Wildcard search*

  Specifies whether a wildcard matching will be used. In wildcard search the asterisk ('*') matches a sequence of zero or more characters of any kind, e.g. whether the search text *dist*ution* will find the text *distribution* or not.

- *Search substructure*

  If this option is set, not only the selected diagram but also its diagram substructure will be searched.

- *Whole word search*

  If this option is set, the search will only find whole words. An whole word is delimited by non alphanumeric characters.

Note:

Only the textual elements that are visible in an MSC will be searched. See "Diagram Options" on page 1676.

Dialog Buttons

When the dialog is first opened, all buttons except *Replace&Search* and *Replace All* are enabled. When *Search*, *Replace&Search* or *Replace All*

is pressed all fields and buttons are disabled except the *Close* button. (The *Close* button changes name during the search to *Abort*.) If a search string is found in an editor, it is selected and all buttons and fields are enabled.

When the first search or replace operation has been applied and control returns to the *Search* dialog, it is possible to perform a new search on the same diagram(s).

- *Search*

  Searches for the search string. An editor is opened when the search string is found. If no text is supplied in the *Search for* field, a warning message appears. Confirming the message box will return control to the *Search* dialog.

- *Replace&Search*

  Replaces the current match with the replace string and searches for the next match. An editor is opened when the search string is found.

- *Replace All*

  Replaces all occurrences of the search string with the replace string. No editors are opened. However, diagrams with replaced text are loaded in the editor and gets a “dirty” state in the Organizer.

  Text in reference symbols and in kernel headings are not replaced.

- *Close/Abort*

  The *Close* button closes the *Search* dialog. The *Close* button is temporarily renamed to *Abort* during search in diagrams. If *Abort* is pressed, the current search is stopped as soon as possible (when a new file is to be searched). After an *Abort*, the dialog remains on screen, ready for new input.

Search Restrictions

All data in the editors that affects the diagram structure maintained by the Organizer (primarily reference symbols and kernel headings in the SDL Editor) is regarded as **read only** during the search operation. That is, they are not affected by the search.

Externally editing (i.e. through means other than using replace) of a diagram during a search operation completely resets the search, i.e. the next search starts from the first diagram.

The search may fail if dialogs are opened in the editor during the search. In this case the editor blocks the continuation of the search process. To continue the search process, the editor dialog must be closed.

The Organizer's data is locked during the search process. This is normally not noticed since the *Search* dialog is modal, but the SDL Editor needs to access that data to perform operations affecting the diagram structure. The Organizer will deny the editor's requests to modify the data structure. The duration of the search process is the period of time during which the *Search* dialog is visible in the Organizer.

Fast Search

Fast search is invoked by pressing Ctrl+F on the keyboard.

Fast search behaves like normal search, except that:

- No dialog is used.
- The replace functionality can not be used.
- Consider case is always off.
- Wildcard search is always off.
- Search substructure is always on.

When Fast search is invoked, the message area displays the text that will be searched for. Initially, the text is *Search for:*. When you type on the keyboard with the mouse pointer over the drawing area, the characters will turn up in the message area. When you have typed the text pattern to search for, press enter or return to start the search operation.

The same search operation as for normal Search is used. If the search operation finds a match, you can search for another match by pressing enter or return once more in the Organizer window.

To finish the Fast search operation, click in the drawing area or select another operation.

The next time Fast search is invoked with Ctrl+F, the search text that was used the last time is proposed as a search text once more. To use it, press enter or return. To use another search text, press Ctrl+F once more or the delete key several times, to erase the search text.

Spelling > Comments

Check the spelling of comments in selected diagrams. Comments can be either comment symbols or /* C-style comments */

> **Note:**
> This command only works if the spelling checker *ispell* has been installed on your computer, and the preference SDT*ISpellCommand correctly identifies the *ispell* executable. *ispell* can be found on the internet and uses the same kind of license as the *emacs* text editor.

For spelling errors, the Spelling dialog appears, with the following possibilities:

- *Word replacement text field*. Initially, the word from the diagram is presented here. One way to correct the spelling mistake is to correct the spelling in this text field and press the *change* button.
- *List of suggestions*. For most words, the spelling checker will present possible corrections here. Click on a word in this list to update the *word replacement text field*. Double-click on a word to update the diagram.
- *Add button*. Add the word currently in the *word replacement text field* to your personal dictionary.
- *Change button*. Update the diagram with the word from the *word replacement text field* and find the next spelling error.
- *Ignore button*. Ignore the current spelling error and find the next spelling error.
- *Ignore All button*. Ignore all occurrences of the current spelling error during this session and find the next spelling error. For more “permanent ignorance”, use the *Add button*.

Spelling > All Text

Works in the same way as *Spelling > Comments*, but all text is checked instead of just comment texts.

Change Bars

This menu choice is used to control the usage of change bars in SDL diagrams. A dialog with two options is opened:

- *Reset change bars for selected SDL diagrams*

  This option is used to reset change bars in SDL diagrams that fall under the selection in the Organizer. Note that removing a change bar makes the diagram dirty, i.e. change bars cannot be removed from diagram files where you do not have write access. If this option is on, a second dialog will be opened telling you about the diagrams that the operation will be applied upon.

- *Create change bars when SDL changes occur*

  This option is used to decide if change bars should be created when an SDL diagram is edited. This option is saved in the system file and is valid for all SDL diagrams in the system. The preference *ChangeBars* is used as default for new SDL systems. See "ChangeBars" on page 249 in chapter 3, *The Preference Manager*.

Compare > SDL Diagrams

This menu choice compares the contents of SDL diagram file pairs. A diagram file pair is constructed by matching an SDL diagram file loaded into the Organizer with an SDL diagram file with the same name, but in a different directory.

> **Note:**
>
> There is a similar operation in the SDL Editor for comparing one SDL diagram pair at a time. See "Compare Diagrams" on page 2027 in chapter 43, *Using the SDL Editor*.

The menu choice is dimmed if there are no SDL diagrams in the Organizer.

In the same way as for the *Search* menu choice, the compare operation is limited by the selection in the Organizer. Only SDL diagrams within the scope of the selection will be considered for the compare operation.

When the Compare SDL Diagrams menu choice is invoked, the compare SDL diagrams setup dialog appears.

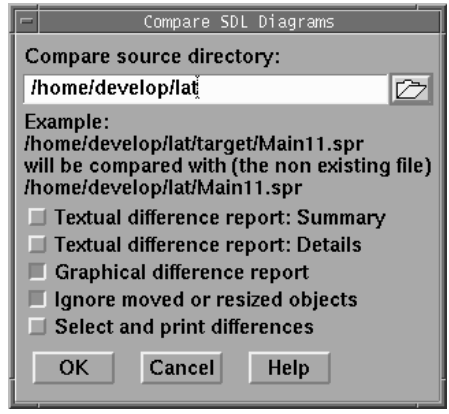


Figure 46: The Compare SDL Diagrams Setup dialog

In the compare SDL diagrams setup dialog, the following input parameters to the compare operation can be specified:

- *Compare source directory*

  The directory specified here is used instead of the normal source directory to find SDL diagrams to compare the SDL diagrams in the Organizer with.

  – For SDL diagrams in the Organizer that are relative to the normal source directory, the relative part of the file path is kept, but the part of the file path that matches the normal source directory is replaced with a part of a path constructed from the diff source directory instead.

    For example, if the source directory is `/home/lat` (`E:\home\lat` **in Windows**) and the diff source directory is `/opt/home/gn` (`E:\opt\home\gn`), then the Organizer file `/home/lat/demo/x.ssy` (`E:\home\lat\demo\x.ssy`) will be compared with `/opt/home/gn/demo/x.ssy` (`E:\opt\home\gn\demo\x.ssy`), if that file exists.

  – For SDL diagrams in the Organizer that are not relative to the normal source directory, the directory part of the file path is replaced with diff source directory.

For example, if the source directory is `/home/lat` (`E:\home\lat` **in Windows**) and the diff source directory is `/opt/home/gn` (`E:\opt\home\gn`), then the Organizer file `/usr/local/lat/y.ssy` (`E:\usr\local\lat\y.ssy`) will be compared with `/opt/home/gn/y.ssy` (`E:\opt\home\gn\y.ssy`), if that file exists.

- *Textual difference report: Summary*

  When this option is chosen, a summary of found differences is printed for each diagram pair that is compared. If more than one diagram pair is compared, a summary for all compared diagram pairs is also printed. The textual difference report is printed in the Organizer Log window.

- *Textual difference report: Details*

  When this option is chosen, detailed information about every found difference is printed as a textual report in the Organizer Log window.

- *Graphical difference report*

  When this option is chosen, differences are shown in the SDL Editor, one difference group at a time.

- *Ignore moved or resized objects*

  When this option is chosen, the compare operation tries to ignore reporting differences only caused by moved or resized symbols by using symbol ids. This option should only be used when comparing two versions of the same original diagram file: moved symbols cannot be accurately detected by using symbol ids when comparing two versions of a diagram that are both built from scratch.

- *Select and print differences*

  When this option is chosen, pages from the compared diagram versions are printed, but only those pages that contain differences.

When the OK button in the dialog is pressed, the Organizer checks if a matching diagram can be found for all SDL diagrams in the operation. If that is not the case, a dialog appears to inform about this fact.

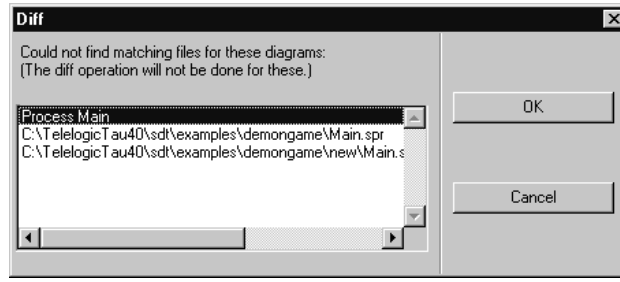


Figure 47: The Compare SDL Diagram missing files dialog

Finally, the real compare operation starts in the SDL Editor. Read more about this in "Compare Diagrams" on page 2027 in chapter 43, *Using the SDL Editor*.

Compare > MSC Diagrams

This menu choice compares the contents of MSC diagram file pairs in the same way as "Compare > SDL Diagrams" on page 148 does. Read more about this in "Compare Diagrams" on page 1690 in chapter 39, *Using Diagram Editors*.

Compare > HMSC Diagrams

This menu choice compares the contents of HMSC diagram file pairs in the same way as "Compare > SDL Diagrams" on page 148 does. Read more about this in "Compare Diagrams" on page 1690 in chapter 39, *Using Diagram Editors*.

Merge > SDL Diagrams

This menu choice compares the contents of SDL diagram file pairs in the same way as "Compare > SDL Diagrams" on page 148 does. The main difference is that this menu choice gives the possibility to merge differences: For each diagram pair, a new merge result diagram is created. Read more about how the actual merge is performed in "Merge Diagrams" on page 2028 in chapter 43, *Using the SDL Editor*.

Merge > MSC Diagrams

This menu choice compares the contents of MSC diagram file pairs in the same way as "Compare > MSC Diagrams" on page 151 does. The

main difference is that this menu choice gives the possibility to merge differences: For each diagram pair, a new merge result diagram is created. Read more about how the actual merge is performed in "Merge Diagrams" on page 1690 in chapter 39, *Using Diagram Editors*.

Merge > HMSC Diagrams

This menu choice compares the contents of SDL diagram file pairs in the same way as "Compare > HMSC Diagrams" on page 151 does. The main difference is that this menu choice gives the possibility to merge differences: For each diagram pair, a new merge result diagram is created. Read more about how the actual merge is performed in "Merge Diagrams" on page 1690 in chapter 39, *Using Diagram Editors*.

Split

This menu choice is used to split one SDL diagram into two SDL diagrams. This menu choice can be applied several times to split one SDL diagram into several parts. This menu choice is, together with the menu choice *Join*, useful in situations where several people have to work simultaneously on one SDL diagram. The SDL diagram file is partitioned into several SDL diagram files and each person is given one part to work on.

To split an SDL diagram consisting of several pages, select the diagram symbol in the Organizer and select the *Split* menu choice. (It is also possible to select a page symbol associated with the diagram that should be split.)

A dialog appears with a list of all pages but the first one in the diagram, see Figure 48. Select a page to define how the SDL diagram should be split. All pages before the selected page will end up in the first SDL diagram part. The selected page and all pages after the selected page will end up in the second SDL diagram part.

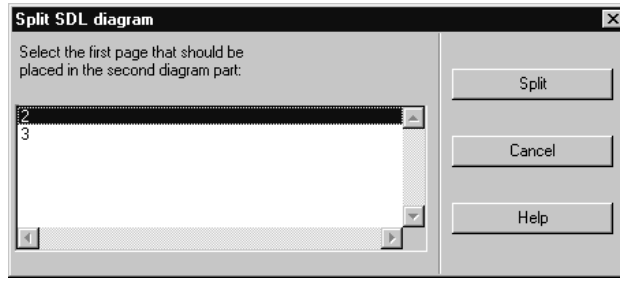


Figure 48: The first Split dialog

The Split button in the first Split dialog closes the dialog and brings up the second Split dialog, as shown in Figure 49. The second Split dialog is used to specify the files that the two resulting SDL diagram parts should be saved in.

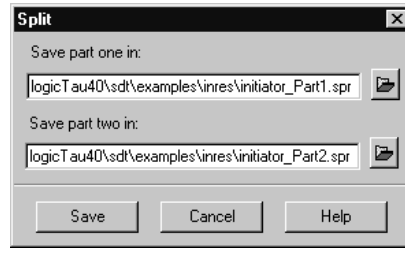


Figure 49: The second Split dialog

When the second Split dialog is closed with the *Save* button, the split operation is performed: The SDL Editor is loaded with the SDL diagram that should be split. Two new smaller SDL diagrams are created in the SDL Editor and saved under the file names specified in the second Split dialog.

The visible result of the Split operation is that the SDL Editor contains at least the diagram that was split and the resulting SDL diagram parts. Note that no additional symbols are created in the Organizer. The SDL diagram parts are accessed by opening the SDL diagram part files in the SDL Editor with the *Open* menu choice. It is of course possible to manually add symbols for the SDL diagram parts in the Organizer with for instance the *Add Existing* menu choice.

Join

This menu choice is used to join two SDL diagrams of the same type into one SDL diagram. This menu choice is, together with the *Split* menu choice, useful when several people have to work on the same SDL diagram at the same time.

Note that it is not necessary to do a Split before doing a Join. One way to work in parallel on the same diagram is to let one designer work on the existing SDL diagram, while another designer creates new pages destined for the same SDL diagram, in a new SDL diagram with the same type as the existing SDL diagram. A join operation when the parallel work is finished puts the new SDL pages in the correct SDL diagram.

The resulting SDL diagram is produced by copying the complete first diagram part and merging/joining pages from the second diagram part by copy and paste. This means that information that is common for all pages in an SDL diagram is taken from the first diagram part. This includes:

- The diagram name and any other information in the header.
- The information in the extended heading symbol.
- The information in the use text symbol.

Another consequence of this way of working is that duplicate page names and reference symbol names emerging from the second diagram part are changed by the join operation to make them unique. Auto-numbered pages from the second diagram part will also have their names changed.

When the *Join* menu choice is invoked, the Join dialog appears, see Figure 50.

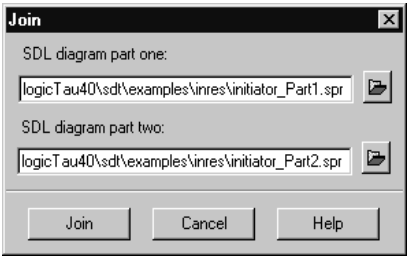


Figure 50: The Join dialog

In the dialog, specify the names of the two SDL diagram files that should be merged. Pressing the *Join* button will close the dialog and start the Join operation.

The result of the Join operation is that three diagrams will be loaded in the SDL Editor; the two SDL diagram parts that act as input to the Join operation and a new and unsaved SDL diagram that contains all the pages from both the input diagrams. The first thing that you will normally do after a Join operation is to save the new SDL diagram in a file. After that, it might be appropriate to check the new SDL diagram in the context of the SDL system with the Analyzer.

Compare State Machines

This menu choice is used to compare state machines with other state machines. This menu choice makes it possible to keep state machines expressed as a group of state charts consistent with state machines expressed as processes in an SDL system.

Before selecting this menu choice, make sure that the information sources (SDL systems and/or state charts) are selected directly or indirectly. For instance, to compare the only SDL system in the Organizer with all state charts in the Organizer, make sure that there is no selection in the Organizer by clicking in the background. The Organizer will interpret this as “everything is selected”.

When you select this menu choice, a dialog is displayed where you can:

- Specify the two groups of state machines that should be compared. A group of state machines is either the state machines in an SDL system or a group of state charts. It is only possible to choose among the state machines that were selected in the Organizer. Use the *Select* buttons to select individual SDL systems or state charts.
- Specify the two state overview information files that will be generated and compared.
- Specify if SDL procedure calls and/or state chart call actions should be compared.
- Specify if SDL outsignals and/or state chart send events should be compared.

When the dialog is closed, two state overview information files are generated. A state overview describes a state machine in a normalized form:

- For an SDL system, diagram type inheritance and diagram type instantiation have been removed.
- For a state chart, state hierarchies (states in states) have been removed. The same rules as when converting a state chart to SDL are used, for more information, see "Converting State Charts to SDL" on page 1702 in chapter 39, *Using Diagram Editors*.

The compare operation compares the two generated state overview information files in the following way:

- State machines are matched by their names. For instance, SDL process A is matching state chart A.
- States are matched by their names.
- Transitions are matched by a combination of from-state name, in-signal name and to-state name

> **Note:**
>
> The compare operation does not compare all the details for a transition. For instance, the conditional expression in an SDL decision symbol is not compared with the guard condition in a state chart transition.

If the compare operation finds anything that does not match, this is reported in the Organizer Log. You use the Organizer Log quick button *Show Error* to navigate to the SDL diagram or state chart with an entity that did not match anything in the other group of state machines.

Simulator Test > New Simulator

This menu choice is used to execute test cases in the simulator for an SDL system. Test cases can be described either as MSCs or as simulator UI input scripts (*.cui). A description of how to express MSCs in this context can be found in *Using MSCs as test cases*.

When this menu choice is invoked, a dialog appears with a list of all MSCs and input scripts. In the dialog, it is possible to select test cases that should be executed. The selection in the Organizer decides:

- primarily the SDL system to be used,
- but also the default test case selection in the dialog.

When the dialog is closed, an SDL simulator is generated and each selected test case is run. When all test cases have been executed, the Organizer Log will contain a one-line summary for each test case, with information about if the test case passed or failed. For test case failures, the MSC editor will pop up, showing the symbol in the MSC that failed (this happens only if the test case was expressed as an MSC). To be able to examine test case failures in detail, run each test case that fails separately, because then the MSC editor will show the place of the MSC test case failure and the textual output from the simulator will contain information about the test case failure.

Using MSCs as test cases

MSC test cases are MSCs written in a special way. MSC test cases are high-level test cases that are auto-converted to low-level simulator UI input script test cases before they are executed.

When MSCs are used as test cases, you can:

- send in a signal to the system and check that the expected signal is received as a response from the system.
- check that a process instance is in a certain state.
- check that a process instance variable has a certain value.
- check if a process instance exists.
- ...

Figure 51 illustrates MSC test case building blocks that are described in the text below.

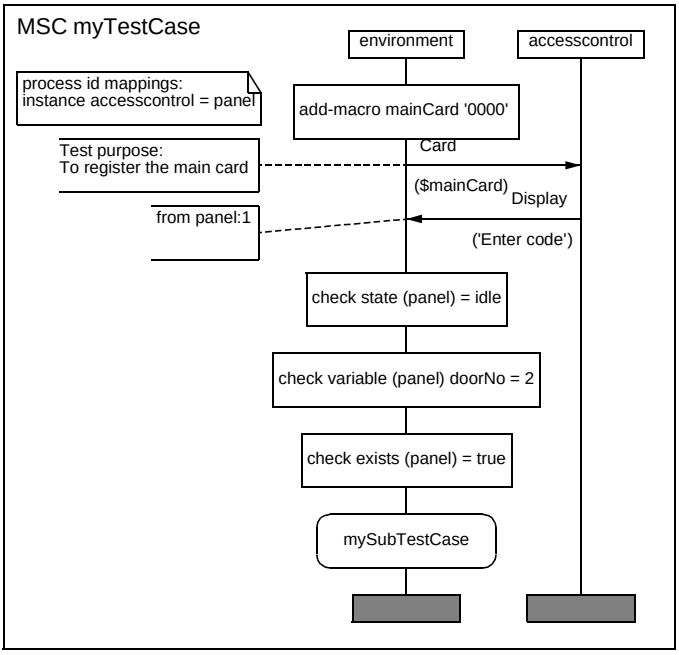


Figure 51 MSC test case building blocks

MSC instance symbols are used in the following way in MSC test cases:

- Use an MSC instance symbol called *environment* or *env_1* or something similar as an instance to send and receive signals to and from the SDL system.

  By using an .itt file other instances than environments can be mapped to act as an environment during testing. See "Mapping Instances to Different Environments" on page 2230 in chapter 49, *The SDL Simulator*.

- Use one or several MSC instance symbols to represent the SDL system. If you use one instance, give it the same name as the SDL system (for instance *accesscontrol*). If you use several instances, give each the name of an SDL process instance that you want to send and receive signals to and from (for instance *panel* or *panel:1*).

To send a signal into the system, draw an MSC signal from the environment instance to one of the instances representing the SDL system. Specify the signal name and any parameters that you know about.

To check that a signal is sent out from the system, draw an MSC signal from an instance representing the SDL system to the environment instance. Also here, you should specify the signal name and any parameters. The check is done by matching a text string created from the MSC with the actual textual output from the simulator. You can use the * character (matching any characters) if you for instance do not want to specify all parameters in the MSC.

MSC test scripts can check for unexpected signals sent to the environment. If unexpected (unchecked) signals are detected **before** the **currently** checked signal the test case will fail and a message will be displayed:

```
Unexpected signal(s) to environment arrived before
currently checked signal:
* OUTPUT of DSignal to env:1
* OUTPUT of ASignal to env:1
```

If unexpected (unchecked) signals are detected **after** the **last** checked signal in the script the test case will fail and a message will be displayed:

```
Unexpected signal(s) to environment arrived after
last checked signal:
* OUTPUT of CSignal to env:1
```

Make sure that the script will run sufficiently long after the last checked signal, for the unexpected signals to arrive at the environment and be detected. This can for example be achieved with a finite number of “next-state” commands at the end of the test script. The statistics displayed after the test script has executed have been updated to show unexpected signals as follows:

```
Command statistics:
        checked: 2
        failed: 0
        updated: 0
        unexpected: 2
```

There are several overlapping ways of specifying the process instance a signal should be sent to or from. Two of them involves a mapping table in an MSC text symbol. The text should look something like this (one line with a ‘=’ for each mapping rule):

```
process id mappings:
instance accesscontrol = panel
signal display = panel
```

- First of all, the SDL Suite finds out if there is a comment symbol connected to the signal. If there is, and the text is something like *to panel* or *from panel*, the SDL Suite uses panel as the process instance name (with *:1* added because no *:* with a number was specified).
- Second, the SDL Suite finds out if there is a signal mapping rule matching the current signal name. If there is, the process instance name in that rule is taken.
- Third, the SDL Suite finds out if there is an instance mapping rule matching the instance name of the instance representing the SDL system that is attached to the current signal. If there is, the process instance name in that rule is taken.
- If no process instance name has been found yet, then the MSC instance name of the instance representing the SDL system is taken as the SDL process instance name. If a qualifier exists in the instance kind text this will be added to the name.

It is possible to include any simulator command in an MSC test script by attaching an MSC action symbol to the environment instance, and typing the simulator commands in it. One line for each command.

This can for instance be used to declare macros representing parameter values. If you have done `add-macro myMacroName 5` in an action symbol, you can type `$myMacroName` instead of `5` as the value of a parameter. Macros can be defined in a separate MSC and used in normal MSC test cases. Just make sure that the macro MSC is executed before the normal MSC test cases when you do a simulator test.

There are three textual shortcuts that can be used in an action symbol:

- To check that an SDL process instance is in a certain state:

```
check state (panel) = IDLE
```

- To check the value of a variable in an SDL process instance:

```
check variable (panel) DoorNo = 1
```

- To check if an SDL process instance exists or not:

```
check exists (panel) = false
```

The MSC reference symbol can be attached to the environment instance. Type in a name of a sub MSC test script that should be executed. This is a way to avoid repeating the same information in many places. It can for instance be used in the beginning of a test script to perform common initialization of the system.

Simulator Test > Existing Simulator

Same as "Simulator Test > New Simulator" on page 156, except that an already created simulator is used. Before the normal simulator test dialog (used to specify test cases to execute), a file selection dialog appears, where a simulator executable can be specified.

Editors > Deployment Editor

Adds a Deployment diagram symbol to the Organizer view, and starts the Deployment Editor. The symbol is added at the same place as when using the *Add New* command.

Editors > HMSC Editor

Adds a HMSC diagram symbol to the Organizer view, and starts the HMSC Editor. The symbol is added at the same place as when using the *Add New* command.

Editors > MSC Editor

Adds an MSC diagram symbol to the Organizer view, and starts the MSC Editor. The symbol is added at the same place as when using the *Add New* command.

Editors > OM Editor

Adds an Object Model diagram symbol to the Organizer view, and starts the OM Editor. The symbol is added at the same place as when using the *Add New* command.

Editors > SDL Editor

Adds an SDL System diagram symbol to the Organizer view, and starts the SDL Editor. The symbol is added at the same place as when using the *Add New* command.

Editors > State Chart Editor

Adds a State Chart diagram symbol to the Organizer view, and starts the State Chart Editor. The symbol is added at the same place as when using the *Add New* command.

Editors > Text Editor

Adds a Plain Text document symbol to the Organizer view, and starts the Text Editor. The symbol is added at the same place as when using the *Add New* command.

Editors > TTCN Browser

Adds a TTCN module symbol to the Organizer view, and starts the TTCN Browser. The symbol is added at the same place as when using the *Add New* command.

SDL > Type Viewer

This menu choice starts the Type Viewer. It is dimmed if there is no SDL diagram in the Organizer, or if the preference SDT*StartInformationServer is set to false. Only one instance of the Type Viewer exists. If the Type Viewer has already been started, its window is raised.

If a Referenced Diagram Type icon, an Instance Diagram icon or a Dashed diagram icon is selected in the Organizer, the Type Viewer selects the corresponding symbol when this menu choice is used.

The Type Viewer is described in chapter 45, *The SDL Type Viewer*.

SDL > Coverage Viewer

This menu choice starts a Coverage Viewer. A new instance of the Coverage Viewer is started each time this command is selected.

The Coverage Viewer is described in chapter 47, *The SDL Coverage Viewer*.

SDL > Index Viewer

This menu choice starts an Index Viewer. A new instance of the Index Viewer is started each time this command is selected.

The Index Viewer is described in chapter 46, *The SDL Index Viewer*.

SDL > Simulator UI

This menu choice starts a new, empty Simulator UI. Several Simulator UI's may exist at the same time.

The Simulator UI is described in "Graphical User Interface" on page 2199 in chapter 49, *The SDL Simulator*.

SDL > Explorer UI

This menu choice starts a new, empty Explorer UI. Several Explorer UI's may exist at the same time.

The Explorer UI is described in "Graphical User Interface" on page 2349 in chapter 52, *The SDL Explorer*.

SDL > Target Tester UI

This menu choice starts a new, empty SDL Target Tester UI. The SDL Target Tester UI is described in "Graphical User Interface" on page 3656 in chapter 67, *The SDL Target Tester*.

TTCN > Find Table

This menu choice invokes the find table operation on a selected TTCN system. **In Windows**, this functionality is not available and the menu choice is dimmed.

For more information about the find table operation **on UNIX**, see "TTCN Suite Preprocessor" on page 1205 in chapter 26, *Analyzing TTCN Documents (on UNIX)*. For more information about finding tables **in Windows**, see "Finding and Sorting Tables" on page 1282 in chapter 30, *Editing TTCN Documents (in Windows)*.

TTCN > Access

This menu choice starts a TTCN Access application for a selected TTCN document/system.

For more information, see chapter 21, *TTCN Access*.

TTCN > Simulator UI

This menu choice starts a TTCN simulator UI.

For more information about this, see chapter 28, *The SDL and TTCN Integrated Simulator (U)* or chapter 33, *The SDL and TTCN Integrated Simulator (W)*.

Preference Manager

This menu choice starts the Preference Manager. Only one instance of the Preference Manager exists. If the Preference Manager has already been started, its window is raised.

The Preference Manager is described in chapter 3, *The Preference Manager*.

Bookmarks Menu

The *Bookmarks* menu contains the following menu choices:

- *Add Bookmark*
- *Edit Bookmarks*
- *More Bookmarks*

Add Bookmark

This menu choice opens a dialog where a new bookmark can be created. The information needed is the *location* (a URL or a SDTREF), and a *name* for the bookmark. If desired, a *systemfile* can be specified that will be loaded together with the SDTREF.

Edit Bookmarks

Edit Bookmarks will open a dialog with a list with all bookmarks. After selecting a bookmark the buttons will have the following effect:

- *OK* - all changes made willbe saved and the dialog will close.
- *Open* - the selected bookmark will be opened
- *Edit* - provides a dialog with the information form the selected bookmark, where the information can be edited.
- *Remove* - removes the selected bookmark from the list.
- *Cancel* - all alteration made will be disregarded and the dialog will be closed.

More Bookmarks

This choice will appear if more than 25 bookmarks are present. The dialog appearing is the same as in the *Edit Bookmark* menu choice. Selecting the desired bookmark and clicking on the *Open* button will open the bookmark.

Help Menu

For more information about *Help* menus, see "Help Menu" on page 15 in chapter 1, *User Interface and Basic Operations*. Two of the Organizer *Help* menu choices are described in more detail below.

About All

This menu choice starts an operation that presents version information about the individual tools in your SDL Suite configuration. The information is presented in the Organizer Log window. The produced information might look like this:

```
About All. Version information:
Help Tool              Version 6.3.0
Link Manager           Version 6.3.0
MSC Editor             Version 6.3.0
OM InfoServer          Version 6.3.0
OM/SC/HMSC/DP Editor   Version 6.3.0
Organizer              Version 6.3.0
Preference Manager     Version 6.3.0
SDL Coverage Viewer    Version 6.3.0
SDL Editor             Version 6.3.0
SDL Index Viewer       Version 6.3.0
SDL Type Viewer        Version 6.3.0
SDT Welcome Window     Version 6.3.0
Text Editor            Version 6.3.0

About All. Additional version and kernel
information:
Information Server version 6.3.0
SDT Analyzer
SDT Analyzer 6.3.0
SDT CPP2SDL 6.3.0
ASN.1 Analyzer 6.3.0
SDL Targeting Expert: Version 6.3.0

Simulation kernel:
  2 lrwxrwxrwx 1 nnn  sdl  66 May 5 16:05 sctworld.o
RealTimeSimulation kernel:
  2 lrwxrwxrwx 1 nnn  sdl  68 May 5 16:05 sctworld.o
...
```

License Information

Opens a dialog with license information for all SDL Suite and TTCN Suite tools.

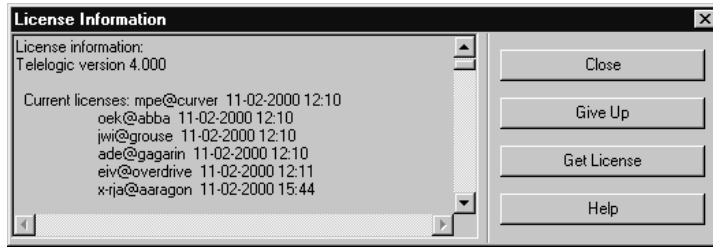


Figure 52: The License Information dialog

The dialog contains the following:

- The names of the tools that are included in SDL Suite and TTCN Suite and that are separately licensed.

  Each name consists of the prefix SDT or ITEX, followed by a hyphen and an abbreviated name for the tool. See the table below for a reference to these abbreviations.

Abbreviation	**Tool(s)**
Telelogic	Organizer
SDT-Base	SDL Editor, SDL Analyzer and SDL Viewers.
SDT-OME	Object Model Editor and State Chart Editor
SDT-MSCE	Message Sequence Chart Editor and High-Level Message Sequence Chart Editor
SDT-Cbasic	Cbasic SDL to C Compiler
SDT-Cadvanced	Cadvanced SDL to C Compiler
SDT-Cmicro	Cmicro SDL to C Compiler
SDT-Cmicro-Bodybuilder	Cmicro BodyBuilder
SDT-Cmicro-Tester	SDL Target Tester UI
SDT-X[a] **(UNIX only)**	<Configuration dependent>

Abbreviation	Tool(s)
SDT-Explorer	Explorer Library
SDT-Simulator	Simulator Library
SDT-Application	Application Library
SDT-Performance	Performance Simulation Library
SDT-TTCN-Link	Interactive link to TTCN Suite
ITEX-Base	TTCN Browser, TTCN Table Editor and TTCN Analyzer
ITEX-Access	TTCN Suite Access API Library
ITEX-Simulator	TTCN Suite Simulator
ITEX-C-Code-Generator	TTCN Suite C Code Generator
SDT-Author	Package licenses, see "Description of packages" on page 59 in chapter 5, *Licensing Management*
SDT-ModelBuilder	
SDT-CodeBuilderCMicro	
SDT-CodeBuilderCAdvanced	

a. The SDT-X is a generic name that allows to introduce new code generators that are under development. In "normal" installations, it has no meaning.

- **On UNIX**, beneath each tool, the following information is displayed:
  - Current licenses, along with the name of the host computer, the user identity and the time the license was checked out
  - Remaining available licenses
  - Total available licenses
  - If a license has expired, the information would be replaced by:

```
SDT-<ToolName>
  Expired: <Date>
```

- A *Give Up License* button which allows you to release the interactive licenses in your current SDL Suite session.

This will disable licensed tools without closing them, making the licenses available to other users. You can reclaim these licenses at a later time, if there are corresponding licenses available, by using the *Get License* button, allowing you to continue working in the licensed applications.

- A *Get License* button which allows you to reclaim licenses released by either the use of the *Give Up License* button or the automatic release of licenses.

  Licenses will be automatically released if the idle time limit, set in the preference LicenseTimeout, is exceeded. For more information, see "LicenseTimeout" on page 244 in chapter 3, *The Preference Manager*.

  The reclamation of licenses is limited in the sense that it will only attempt to reclaim lost licenses once, implying that you will have to save your work in and restart the tools for which licenses could not be obtained.

- A *Configure License* button which allows you to configure package licenses. See "Set up and change a Package license" on page 60 in chapter 5, *Licensing Management*.

Shortcuts

There are several ways to invoke an operation in the Organizer. An operation is usually available in the menu bar and in at least one pop-up menu. Menus can be operated via the keyboard (keyboard navigation). Several operations are also available in the tool bar or via keyboard accelerators. For instance, the Print operation can be invoked by:

- Choosing the menu choice in the *File* menu in the menu bar with the mouse.
- Choosing the menu choice in the *File* menu in the menu bar with keyboard navigation. (`<Alt+F> P`)
- Using the keyboard accelerator `<Ctrl+P>`.
- Pressing the *Print* quick button.
- Choosing the menu choice in the *File* sub menu of the background pop-up menu. (The *Print* menu choice is also available in several other pop-up menus.)

The sections below are “Pop-Up Menus” on page 170, “Keyboard Accelerators” on page 178 and “Quick Buttons” on page 180.

Pop-Up Menus

This section describes pop-up menus for different types of icons. The operations available depend on the type of file that icon represents.

The following tables lists the menu choices in the pop-up menus and a reference to the corresponding menu choice in the menu bar.

On SDL Diagrams (but not SDL Overview Diagrams)

Print Selected	“Print > Selected” on page 70
Set Directories	“Set Directories” on page 71
Edit > Edit	“Edit” on page 86
Edit > Remove	“Remove” on page 91
Edit > Connect	“Connect” on page 92
Edit > Disconnect	“Disconnect” on page 96

Edit > CM Group	“Configuration > Group File” on page 97
Edit > CM Update	“Configuration > Update” on page 98
Edit > Associate	“Associate” on page 98
Edit > Update Headings	“Update Headings” on page 100
View > Expand	“Expand” on page 103
View > Expand Substructure	“Expand Substructure” on page 104
View > Collapse	“Collapse” on page 104
View > Show Sub Symbols	“Show Sub Symbols” on page 105
View > Hide	“Hide” on page 106
Generate > Analyze	“Analyze” on page 112
Generate > Make	“Make” on page 120
Generate > Stop Analyze/Make	“Stop Analyze/Make (UNIX only)” on page 127
Generate > Generate SDL Overview	“SDL Overview” on page 128
Generate > Convert to PR	“Convert to PR/MP” on page 131
Generate > Convert to GR	“Convert to GR” on page 133
Generate > Edit Separation	“Edit Separation” on page 137
Generate > Dependencies	“Dependencies” on page 138
Search	“Search” on page 142
Change Bars	“Change Bars” on page 147
Compare SDL Diagrams	“Compare > SDL Diagrams” on page 148
Merge SDL Diagrams	“Merge > SDL Diagrams” on page 151
Create Endpoint	“Link > Create Endpoint” on page 142
Traverse Link	“Link > Traverse” on page 142

Clear Endpoint	"Link > Clear Endpoint" on page 142
Type Viewer	**(On SDL type diagrams only)** "SDL > Type Viewer" on page 162. Opens the Type Viewer with the type diagram selected.

On MSC and SDL Overview Diagrams

Print Selected	"Print > Selected" on page 70
Set Directories	"Set Directories" on page 71
Edit > Edit	"Edit" on page 86
Edit > Remove	"Remove" on page 91
Edit > Connect	"Connect" on page 92
Edit > Disconnect	"Disconnect" on page 96
Edit > CM Group	"Configuration > Group File" on page 97
Edit > CM Update	"Configuration > Update" on page 98
Edit > Associate	"Associate" on page 98
Hide	"Hide" on page 106
Dependencies	"Dependencies" on page 138
Search	"Search" on page 142
Change Bars	"Change Bars" on page 147
Create Endpoint	"Link > Create Endpoint" on page 142
Traverse Link	"Link > Traverse" on page 142
Clear Endpoint	"Link > Clear Endpoint" on page 142

On Instance and Dashed Diagrams

Edit	"Edit" on page 86.
Type Viewer	"SDL > Type Viewer" on page 162. Opens the Type Viewer with the diagram selected.

Hide	"Hide" on page 106

On Pages

Print Selected	"Print > Selected" on page 70
Edit	"Edit" on page 86.
Hide	"Hide" on page 106

On Associations

Associations have the same pop-up menu as the associated document. All menu choices operate on the associated document, except *Hide* and *Remove* that operate on the association symbol itself.

On HMSC, Object Model and State Chart Diagrams

Print Selected	"Print > Selected" on page 70
Set Directories	"Set Directories" on page 71
Edit > Edit	"Edit" on page 86
Edit > Remove	"Remove" on page 91
Edit > Connect	"Connect" on page 92
Edit > Disconnect	"Disconnect" on page 96
Edit > CM Group	"Configuration > Group File" on page 97
Edit > CM Update	"Configuration > Update" on page 98
Edit > Associate	"Associate" on page 98
Edit > Update Headings	"Update Headings" on page 100
View > Expand	"Expand" on page 103
View > Expand Substructure	"Expand Substructure" on page 104
View > Collapse	"Collapse" on page 104
View > Show Sub Symbols	"Show Sub Symbols" on page 105
View > Hide	"Hide" on page 106

Dependencies	“Dependencies” on page 138
Search	“Search” on page 142
Change Bars	“Change Bars” on page 147
Create Endpoint	“Link > Create Endpoint” on page 142
Traverse Link	“Link > Traverse” on page 142
Clear Endpoint	“Link > Clear Endpoint” on page 142

On Modules

Print Selected	“Print > Selected” on page 70
Edit	“Edit” on page 86
Remove	“Remove” on page 91
CM Group	“Configuration > Group File” on page 97
CM Update	“Configuration > Update” on page 98
View > Expand	“Expand” on page 103
View > Expand Substructure	“Expand Substructure” on page 104
View > Collapse	“Collapse” on page 104
View > Show Sub Symbols	“Show Sub Symbols” on page 105
Merge ASN.1	“Merge ASN.1” on page 139
Search	“Search” on page 142
Change Bars	“Change Bars” on page 147
Compare SDL Diagrams	“Compare > SDL Diagrams” on page 148
Merge SDL Diagrams	“Merge > SDL Diagrams” on page 151

On Chapters

Print Selected	“Print > Selected” on page 70

Edit	“Edit” on page 86
Remove	“Remove” on page 91
CM Group	“Configuration > Group File” on page 97
Search	“Search” on page 142
Compare SDL Diagrams	“Compare > SDL Diagrams” on page 148

On More Symbols

Show Sub Symbols	“Show Sub Symbols” on page 105

On System File

CM Group	“Configuration > Group File” on page 97
CM Update	“Configuration > Update” on page 98

On Link File

Link Manager	“Link > Link Manager” on page 142

On Source and Target Directories

Set Directories	“Set Directories” on page 71

On the Background

Expand Substructure	“Expand Substructure” on page 104
Collapse	“Collapse” on page 104
View Options	“View Options” on page 106
Set Scale	“Set Scale” on page 111
File > New	“New” on page 59
File > Open	“Open” on page 60

File > Save	“Save” on page 62
File > Save As	“Save As” on page 65
File > Print All	“Print > All” on page 70
File > Set Directories	“Set Directories” on page 71
File > PC Drives	“PC Drives” on page 73
File > Compare System	“Compare System” on page 74
File > Merge System	“Merge System” on page 74
File > Import SDL	“Import SDL” on page 80
File > Exit	“Exit” on page 84
Edit > Add New	“Add New” on page 89
Edit > Add Existing	“Add Existing” on page 90
Edit > Paste As	“Paste As” on page 99
Edit > Go to Source	“Go To Source” on page 100
Convert to GR	“Convert to GR” on page 133
Tools > Organizer Log	“Organizer Log” on page 142
Tools > Link Manager	“Link > Link Manager” on page 142
Tools > Search	“Search” on page 142
Tools > Change Bars	“Change Bars” on page 147
Tools > Compare SDL Diagrams	“Compare > SDL Diagrams” on page 148
Tools > Merge SDL Diagrams	“Merge > SDL Diagrams” on page 151
Tools > Type Viewer	“SDL > Type Viewer” on page 162
Tools > Coverage Viewer	“SDL > Coverage Viewer” on page 162
Tools > Index Viewer	“SDL > Index Viewer” on page 162
Tools > Simulator UI	“SDL > Simulator UI” on page 163
Tools > Explorer UI	“SDL > Explorer UI” on page 163
Tools > Target Tester UI	“SDL > Target Tester UI” on page 163

Tools > Preference Manager	"Preference Manager" on page 164
Editors > Deployment Editor	"Editors > Deployment Editor" on page 161
Editors > HMSC Editor	"Editors > HMSC Editor" on page 161
Editors > MSC Editor	"Editors > MSC Editor" on page 161
Editors > OM Editor	"Editors > OM Editor" on page 161
Editors > SDL Editor	"Editors > SDL Editor" on page 161
Editors > State Chart Editor	"Editors > State Chart Editor" on page 162
Editors > Text Editor	"Editors > Text Editor" on page 162
Editors > TTCN Browser	"Editors > TTCN Browser" on page 162
Help > Contents *Help > On Organizer* *Help > On Shortcuts* *Help > New Features* *Help > Latest News* *Help > Index* *Help > Search* *Help > IBM Home Page* *Help > Help Desk* *Help > About Organizer*	For more information about *Help* menu choices, see "Help Menu" on page 15 in chapter 1, *User Interface and Basic Operations*.
Help > License Information	"License Information" on page 166

Keyboard Accelerators

Apart from the general keyboard accelerators, as described in "Keyboard Accelerators" on page 35, the following accelerators can be used in the Organizer:

Accelerator	Reference to corresponding command or quick button
`Ctrl+A`	"Add New" on page 89
`Ctrl+B`	"Connect" on page 92
`Ctrl+E`	"Edit" on page 86
`Ctrl+I`	"Import SDL" on page 80
`Ctrl+K`	"Stop Analyze/Make (UNIX only)" on page 127
`Ctrl+M`	"Make" on page 120
`Ctrl+R`	"Disconnect" on page 96
`Ctrl+T`	Toggles between *Analyze*/*Make* (default) and *Full Analyze*/*Full Make*. A text is presented in the message area to confirm that `<Ctrl+T>` has been pressed: Either "Organizer will now use full analyze/make" or "Organizer will now use normal analyze/make". For more information, see "Normal versus full analyze (make)" on page 114.
`Ctrl+0 (zero)`	"Set Directories" on page 71
`Ctrl+1`	"Organizer Log" on page 142
`Ctrl+2`	"Add Existing" on page 90
`Ctrl+3`	"Associate" on page 98
`Ctrl+4`	"Analyze" on page 112
`Ctrl+5`	"SDL Overview" on page 128
`Ctrl+6`	"SDL > Type Viewer" on page 162
`Ctrl+7`	"SDL > Coverage Viewer" on page 162
`Ctrl+8`	"SDL > Index Viewer" on page 162

Accelerator	Reference to corresponding command or quick button
`Ctrl+9`	“Preference Manager” on page 164
`Del`	“Remove” on page 91. For documents that can not be removed (non-root SDL diagrams), see “Disconnect” on page 96.
`arrow up`	Select the symbol one step up (move the selection)
`Shift+arrow down`	“Move Up” on page 182.
`Ctrl+Shift+ arrow up`	Place the selected symbol first in the Organizer structure.
`arrow down`	Select the symbol one step down (move the selection)
`Shift+arrow down`	“Move Down” on page 182
`Ctrl+Shift+ arrow down`	Place the selected symbol last in the Organizer structure.

Quick Buttons

The following quick buttons are special to the Organizer's Main window. The general quick buttons are described in "General Quick-Buttons" on page 24. Each quick button in the Organizer has a preference parameter that specifies if the button is shown or hidden; see "Organizer Preferences" on page 246. As default, all quick buttons except *New* and *Add Existing* are shown.

New
Creates a new system; see "New" on page 59.

Save
Performs a silent *Save All* of all diagrams including the system file. Does not bring up the *Save* dialog, except if diagrams are modified and unconnected, or no system file exists. Corresponds to the *Save All* button in the Save dialog; see "Save" on page 62.

Print
Brings up the *Print* dialog, see chapter 5, *Printing Documents and Diagrams*.

Analyze
Analyzes the part of the system defined by the selected diagram or the whole system. Brings up a *Save Before* dialog if any diagram is modified and not saved. Does not bring up the *Analyze* dialog. Corresponds to the *Analyze* button in the Analyze dialog. The current Analyze options are used. The same restrictions and checks apply as described in "Analyze" on page 112.

Make
Makes the part of the system defined by the selected diagram or the whole system, i.e. generates code, compiles and links it. Does not bring up the *Make* dialog. Corresponds to the *Make* button in the Make dialog. The current Analyze and Make options are used. The same restrictions and checks apply as described in "Make" on page 120.

Generate SDL Overview
Generates an SDL overview diagram for the selected diagram or the whole system. Does not bring up the *SDL Overview* dialog. Corresponds to the *Generate* button in the *Generate SDL Overview* dialog. The current *Generate SDL Overview* options are used. The same restrictions and checks apply as described in "SDL Overview" on page 128.

Simulate
Generates a simulator for the part of the system defined by the selected diagram or the whole system, and starts a Simulator UI (see "Simulator/Explorer UI Started from Quick Buttons" on page 182). Does not bring up the *Make* dialog if the simulator needs to be regenerated. If a simulation kernel has not been selected in this dialog, the default simulation kernel is used. The default simulation kernel is determined by the preference Organizer*DefaultSimulator. The default value is "SCTADEBCOM" which is the same as using the standard kernel *Simulation* in the Make dialog. The same restrictions and checks apply as described in "Make" on page 120.

Explore
Generates an SDL Explorer for the part of the system defined by the selected diagram or the whole system, and starts an Explorer UI (see "Simulator/Explorer UI Started from Quick Buttons" on page 182). Does not bring up the *Make* dialog if the explorer needs to be regenerated. If a validation kernel has not been selected in this dialog, the default validation kernel is used. The default validation kernel is determined by the preference Organizer*DefaultExplorer. The default value is "SCTAVALIDATOR" which is the same as using the standard kernel *Validation* in the Make dialog. The same restrictions and checks apply as described in "Make" on page 120.

Generate Cross References
Generates a cross reference file for the current system and presents the information in the Index Viewer. An index symbol is added in the Organizer view. Does not bring up the *Analyze* dialog. Corresponds to setting the *Generate a cross reference file* option in this dialog. The same restrictions and checks apply as described in "Analyze" on page 112 and "Semantic analysis" on page 115.

Add New
Adds a new document as root; see "Add New" on page 89.

Add Existing
Adds an existing document as root; see "Add Existing" on page 90.

Move Down
Moves the selected symbol one step down in the Organizer structure. For more information about moving symbols, see "Ordering" on page 55.
Moves down the selected page. For more information about moving pages, see "Move down" on page 2035 in chapter 43, *Using the SDL Editor* and "Move down" on page 1680 in chapter 39, *Using Diagram Editors*.

Move Up
Moves the selected symbol one step up in the Organizer structure. For more information about moving symbols, see "Ordering" on page 55.
Moves up the selected page. For more information about moving pages, see "Move up" on page 2034 in chapter 43, *Using the SDL Editor* and "Move up" on page 1680 in chapter 39, *Using Diagram Editors*.

Toggle Pages
Shows or hides page symbols below diagram symbols in the Organizer structure.

Organizer Log
Opens the Organizer Log window; see "Organizer Log" on page 142.

Simulator/Explorer UI Started from Quick Buttons

When the Simulate quick button is used and if the generated simulator already was loaded in a Simulator UI, the same Simulator UI will be re-used for the newly generated simulator. If no Simulator UI is active, a new Simulator UI will be started. However, if there already are other started Simulator UI's, the user has the choice of selecting one of these to load the generated simulator in.

The following dialog is opened:

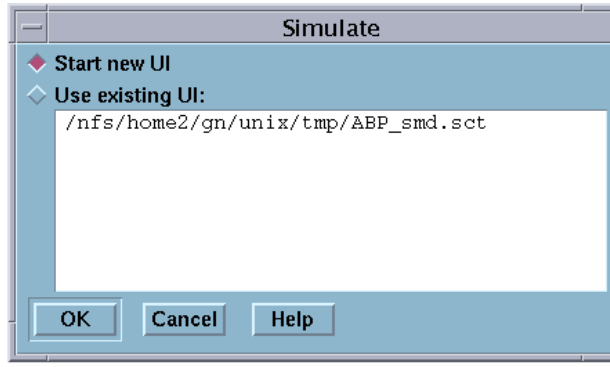


Figure 53: The Simulator/Explorer dialog

- *Start new UI*

  Starts a new Simulator UI and loads the simulator.

- *Use existing UI*

  From the list box, one of the active Simulator UI's can be selected to load the simulator in. If a simulator already was running in the selected Simulator UI, that simulator will be terminated.

The same applies also for an SDL Explorer and the Explorer UI.

Organizer Log Window

The Organizer log window is a sub window to the main window. It works as a console for the SDL Suite and TTCN Suite tools. The window can be visible (raised or iconified) or not visible on the screen. All log information is output to this window independently of whether the window is visible or not.

The window is used in the following situations:

- To log information from the Analyze and Make process.
- To log the activities when an SDL system is imported.
- When files are checked during the *Open* command. Inconsistent files are reported.
- To show status information from the Generate tools.

- Other tools, such as TTCN Suite and tools started with dynamic menus, may also use the window to output textual information.

The window is opened or raised when the menu choice *Organizer Log* is selected from the *Tools* menu, or when something is written to the log. The preference ShowLogLevel controls which output to the window should cause a raise of the window (see chapter 3, *The Preference Manager*).

There is only one Organizer Log window. The Organizer main window is not locked for user input when the Organizer Log window is visible.

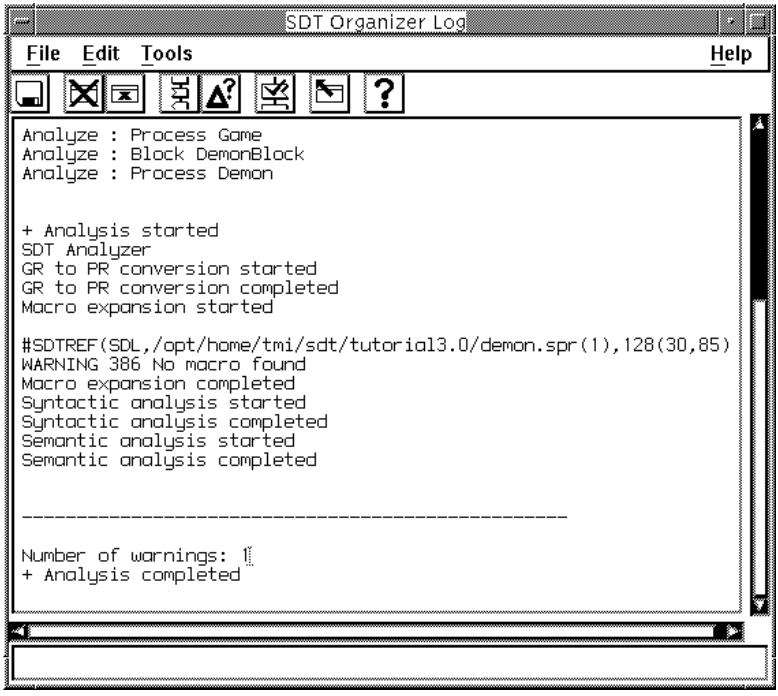


Figure 54: The Organizer Log window

Quick Buttons

The following quick buttons are special to the Organizer Log window. The general quick buttons are described in "General Quick-Buttons" on page 24. Each quick button in the Organizer Log window has a preference parameter that specifies if the button is shown or hidden; see "Organizer Preferences" on page 246 in chapter 3, *The Preference Manager*. All quick buttons in the Organizer Log are shown by default.

Close
Exactly the same as *Close* in the *File* menu.

Clear Log
Exactly the same as *Clear Log* in the *Edit* menu; see "Clear Log" on page 186.

Navigate Down
Move to the next error visible in the Organizer Log. See "Log Options" on page 186 for details.

Navigate Up
Move to the previous error visible in the Organizer Log. See "Log Options" on page 186 for details.

Show Error
Exactly the same as *Show Error* in the *View* menu; see "Show Error" on page 187.

Help on Error
Exactly the same as *Help on Error* in the *View* menu; see "Help on Error" on page 187.

Analyze
Exactly the same as the *Analyze* quick button in the Organizer Main window; see "Analyze" on page 180.

Show Organizer
Exactly the same as *Show Organizer* in the *Tools* menu; see "Show Organizer" on page 187.

File Menu

See "File Menu" on page 8 in chapter 1, *User Interface and Basic Operations*.

Edit Menu

The following menu choices are available in the Organizer log window *Edit* menu:

Copy

This menu choice copies the selected text to the clipboard buffer.

Select All

This menu choice selects all text in the Organizer Log window.

Clear Log

This menu choice clears all text in the Organizer Log window.

Log Options

This menu choice lets you decide how error, warning and information messages from the Analyzer should be handled in the Organizer Log:

- The Filter text field can be used to specify warning messages that should not be shown.
- By pressing the Ignore WARNINGs button and enter a previuosly saved Organizer Log file, all warnings issued by the Analyzer that already exist in the Log file are ignored.
- The navigation level decides if the navigate up and down quick buttons should navigate through:
  - Error messages
  - Error and warning messages
  - All messages

The default value for the Filter text field is taken from the preference Organizer*LogFilter. The default value for the navigation level is taken from the preference Organzier*LogNavigationLevel.

The Ignore WARNINGs button can be used when you repeatedly analyze the same system and only are concerned with warnings issued by new changes in the system. In the Log file any line starting with "#SDTREF" together with the next line will be used to match a warning issued by the Analyzer. The initial number on the second line is always replaced by a wildcard character '*', which means that any number of

characters will match. Additional wildcards can be added manually to tailor the exact behavior of what warnings to be ignored. To make the Log file independent of a specific source directory path, the initial path can be exchanged by a wildcard character.

Example 3 ----------

```
#SDTREF(SDL,*Main1.spr(1),179(80,75),1,1)
1: WARNING 295 Consider adding else answer
```

Tools Menu

The following menu choices are available in the *Tools* menu of the Organizer log window:

Show Organizer

This menu choice raises the Organizer's Main window.

Show Error

This menu choice opens an editor and shows an object/text that corresponds to a selected error/warning. If no text in the log is selected, the menu choice operates on the first error/warning in the log, which becomes selected. If this menu choice is invoked several times without changing the selection manually, the selection is updated automatically to point to the next error/warning before opening the editor and selecting the corresponding object/text.

Help on Error

This menu choice activates the Help tool on the selected error/warning or the next found error/warning. The selection is not updated automatically when selecting this command repeated times (as opposed to *Show Error*, above).

Search

This menu choice opens a dialog that allows you to search for a text in the Organizer log; see "Search" on page 142.

Repeated Search

Repeated search can be invoked without opening the Search dialog by pressing Ctrl+F on the keyboard. This will use the settings of the Search dialog for a new search.

System File

The Organizer maintains a system file that contains a description of all files that are included in the system, together with all settings the user has made to this particular system. The default file name extension for a system file is `.sdt`.

In addition, a system window state file contains information about window positions and sizes, and it uses the extension `.sdt.state`.

Contents of the System File

The contents of the system file summarizes the information that the Organizer manages:

- Source Directory and Target Directory.
- Link file reference (used by the Link Manager) and information about endpoints in the Organizer view.
- All document names are listed, both with logical names and file names. The fact that document names are present means that the Organizer will have a robust recovery when files are manipulated outside the Organizer.
- Associations and dependencies are stored as indexes to other documents for a document.
- The names of all chapters and modules are listed.
- The names of all Control Unit files.

The system file may contain a mapping of directory paths on Windows and UNIX to allow the system to be accessed on both platforms. See "Windows and UNIX File Compatibility" on page 215 for more information.

A number of options are also stored in the system file, but only if they differ from the values in the preference file or if the preference Organizer*AllPreferences is on. The options are:

- Window options as set in the *View Options* and *Set Scale* dialogs
- Analyze options as set in the *Analyze* dialog
- Make options as set in the *Make* dialog
- Print options as set in the *Print > Selected* dialog

In addition, information about collapsed, hidden and selected documents are saved to enable the Organizer to open with the same appearance as when the system file was saved. However, simply changing which documents are collapsed, hidden and selected will not mark the system file as “dirty” in the Organizer.

Format of the System File

The system file is a line-oriented, human-readable text file. The system file is divided into sections. Each section contains lines formatted in the same way. Any lines before the first section are ignored, but saved for the next system file save. This means that it is possible to have user defined comments in the beginning of the file.

Caution!

The sections that describe the diagram and document structure should not be modified by the user unless absolutely necessary. Modifying these sections in a way that errors or inconsistencies are introduced may corrupt the system file.

The file has the following format:

```
<file> ::= <comments> SDT-SYSTEM-6.3(<drives> | $)
           (<dir> | $) (<links> | $)
           (<diagram> | $) (<view> | $) (<analys> | $)
           (<cpp2sdl> | $) (<make> | $) (print | $)

<comments> ::= <comment>*
<comment> ::= Any line of ASCII characters not
beginning with "SDT-SYSTEM-".

<drives> ::= [DRIVES] <drive>*
<drive> ::= <Windows dir> <UNIX dir>

<dir> ::= [SOURCE-TARGET-DIRECTORY] <option>*
<links> ::= [LINKS] <option>*

<diagram> ::= [DIAGRAMS] <dia>*
<dia> ::= <indent> <type> <name>
          (<path> | <fileName> | %unconnected%)
          <viewState> <timeStamp> <separateName>
          <associations> <dependencies>

<view> ::= [VIEWOPTIONS] <option>*
<analys> ::= [ANALYSEROPTIONS] <option>*
<cpp2sdl> ::= [CPP2SDLOPTIONS] <option>*
<make> ::= [MAKEOPTIONS] <option>*
```

<print> ::= [PRINTOPTIONS] <option>*

<Windows dir> ::= <string>
An Windows path. This is either a drive letter and a colon (e.g. C:) or a full path (e.g. C:\SDL). UNC paths can be used (e.g. \\MYHOST\SDL). If you include a trailing backslash you must also include a trailing slash in the corresponding <UNIX dir>.

<UNIX dir> ::= <string>
A UNIX directory path corresponding to the <Windows dir>. If you include a trailing slash you must also include a trailing backslash in the corresponding <Windows dir>.

<indent> ::= <int>
The indentation level. 0 is root, -1 is used for chapters.

<type> ::= <int>
Type of document, module or chapter. If the diagram has some kind of virtuality, the value is factored with a number with the base 100. The value corresponds to the kind of virtuality. These numbers correspond to an enumerated value found in the file sdt.h. If the diagram is an instance, a value of 1000 will be added to the type.

<name> ::= <string>
Logical name of diagram or document.

<path> ::= <string>
File path including a directory.

<fileName> ::= <string>
File name.

<viewState> ::= <integer>
Viewing state, consisting of four weighted booleans for separation, expanded, shown and selected states:
Separation + 2*NotExpanded + 4*NotShown + 8*Selected.

<timeStamp> ::= <integer>
Last time the file was modified, and that the Organizer was aware of. When the Organizer reads the system file, only files with modification dates later than the time stamp are checked for correctness.

<separateName> ::= <string>
Name of separation. Only applicable for units that are separately analyzed.

<associations> ::= <string>
A string of space separated values referencing associated documents, e.g. “1 4 5”. The values index documents in the DIAGRAMS section.

<dependencies> ::= <string>

A string of space separated values referencing documents that this document is depending on, e.g. “1 4 5”. The values index documents in the DIAGRAMS section.

```
<option> ::= (<option-name> = <option-value>)
<option-name> ::= <string>
```

Any option found in the named dialogs.

```
<option-value> ::= <string>
```

Any valid value bound to an option.

If a section is missing entirely from the system file, a warning will be logged, except for the `cpp2sdl` section which is optional and only useful in batch. If no recognizable sections could be found, the file is not a valid system file.

When loading system files, warnings will be registered in the Organizer Log window if any non-recognized option is encountered, but not if an option never was initialized.

The Drives Section

The purpose of the `DRIVES` section in the system file is to achieve file compatibility between UNIX and Windows systems. It specifies the *drive table*; a mapping between Windows and UNIX directory paths. See “Windows and UNIX File Compatibility” on page 215 for more information.

The path format of the current platform is used for directory paths stored in the system file: UNIX path names on UNIX systems (i.e. starting with a slash ‘/’ and directories separated by slashes), and Windows path names on Windows systems (i.e. starting with a drive letter “x:” and directories separated by backslashes ‘\’, or using the UNC format “\\<host>\file”).

Diagrams with file specifications in an incorrect format are marked as Invalid in the Organizer.

When reading the system file on Windows systems, file specifications in UNIX format are converted to Windows format using the mapping in the `DRIVES` section, if possible, including converting slashes to backslashes. When the system file is saved, the file specifications are saved in Windows format, i.e. they are not converted back to UNIX format.

On UNIX systems, file specifications in Windows format are converted to UNIX format using the mapping in the `DRIVES` section, if possible,

including converting backslashes to slashes. When the system file is saved, the file specifications are saved in UNIX format, i.e. they are not converted back to Windows format.

Example 4 ----------

The `DRIVES` section of the system file looks like this:

```
[DRIVES]
C:\TEMP /tmp
\\MYHOST\STORAGE /home/user
```

The file specification `/tmp/a.ssy` in the system file is converted to `C:\TEMP\a.ssy`.

The file specification `/home/user/mydir/a.ssy` is converted to `\\MYHOST\STORAGE\mydir\a.ssy`.

The file specification `/usr/local/dir/a.ssy` is converted to `\usr\local\dir\a.ssy`. Since no drive or host name could be matched, the file will not be found and the diagram will be marked as Invalid in the Organizer.

Options in the System File

A number of sections contain options for the Organizer, representing values that can be set in the Organizer dialogs. If an option in the file is not recognized by the Organizer, it will be ignored, and if an option is not included in the file, a default preference value will be used. A few of the options have no corresponding preference parameters.

The following tables list the Organizer options that are saved in the system file. The user should not normally need to know or change these options in the system file. However, when running the SDL Suite in batch mode **on UNIX**, it may be useful to change some options by editing a system file to be submitted as input to `sdtbatch`. See "Batch Facilities" on page 208 for more information.

The options must appear in the correct section of the system file, but the ordering of options within a section is not important. The options are stored in the system file according to the format:

```
NameOfOption=Value
```

The option names and option values are case insensitive.

Possible and allowed values are not specified; the user should run the Preference Manager in order to obtain a reference to the permitted values.

SOURCE-TARGET-DIRECTORY Section

Option	Default	Corresponding Preference
AbsolutePath	false	*Organizer*AbsolutePath*
SourceDirectory	""	*Organizer*SourceDirectory*
TargetDirectory	""	*Organizer*TargetDirectory*

VIEWOPTIONS Section

Option	Default	Corresponding Preference
Scale	100	*SDT*Scale*
ShowDashed	true	*Organizer*ShowDashed*
ShowDependencies	true	*Organizer*ShowDependencies*
ShowDirectories	true	*Organizer*ShowDirectories*
ShowFileNames	true	*Organizer*ShowFileName*
ShowGroups	true	*Organizer*ShowGroups*
ShowInstances	true	*Organizer*ShowInstances*
ShowLinkFile	false	*Organizer*ShowLinkFile*
ShowLinks	true	*Organizer*ShowLinks*
ShowLongMenus	true	*Organizer*ShowLongMenus*
ShowPages	false	*Organizer*ShowPages*
ShowPermissions	true	*Organizer*ShowPermissions*
ShowSeparators	true	*Organizer*ShowSeparators*
ShowStatusBar	true	*Organizer*Statusbar*
ShowSystemFile	true	*Organizer*ShowSystemFile*

Option	Default	Corresponding Preference
ShowToolBar	true	*Organizer*Toolbar*
ShowTypeName	false	*Organizer*ShowTypeName*
ShowVirtuality	true	*Organizer*ShowVirtuality*
TreeRepresentation	List	*Organizer*TreeRepresentation*

ANALYSEROPTIONS Section

Option	Default	Corresponding Preference
AllowImplicitTypeConv	false	*Organizer*AllowImplicitTypeConv*
ASN1KeywordFile	false	Not available
ASN1KeywordFileName	""	*Organizer*ASN1KeywordFileName*
ASN1Parameter	false	Not available
CaseSensitiveSDL	false	*SDT*CaseSensitive*
CoderBufferInSDL		*Organizer*CoderBufferInSDL*
EchoAnalyzerCommands	false	*Organizer*EchoAnalyzerCommands*
ErrorLimit	30	*Organizer*ErrorLimit*
ExpandPR	false	*Organizer*ExpandPR*
ExpressionLimit	0	*Organizer*ExpressionLimit*
Filter	false	*Organizer*FilterCommand*
FilterCommand	""	*Organizer*FilterCommand*
IgnoreHidden	true	Not available
IncludeOptionalFields	false	*Organizer*IncludeOptionalFields*
InstanceFile	false	Not available
MacroExpansion	false	*Organizer*MacroExpansion*
MissingAnswerValues-Control	true	*Organizer*MissingAnswerValues-Control*
MissingElseControl	true	*Organizer*MissingElseControl*

Option	Default	Corresponding Preference
ParameterMismatchControl	true	*Organizer*ParameterMismatchControl*
ExternalTypeFreeControl	true	*Organizer*ExternalTypeFreeControl*
OptionalParamControl	true	*Organizer*OptionalParamControl*
OutputControl	true	*Organizer*OutputControl*
ReferenceControl	true	*Organizer*ReferenceControl*
SemanticControl	true	*Organizer*SemanticControl*
SyntaxControl	true	*Organizer*SyntaxControl*
TerminateAnalyzer	false	*Organizer*TerminateAnalyzer*
TrailingParamControl	true	*Organizer*TrailingParamControl*
UpperCase	false	*Organizer*UpperCase*
UsageControl	true	*Organizer*UsageControl*
XRef	true	*Organizer*XRef*

CPP2SDLOPTIONS Section, optional section which can be used in batch mode

Option	Default	Corresponding Preference
InputLanguageC	false	Not available
InputLanguageCpp	true	Not available
InputLanguageMicrosoft	false	Not available
InputLanguageGNU	false	Not available
RTTI	false	Not available
RecognizeSDLsorts	false	Not available
ObjectSlicing	false	Not available
Preprocessor	""	Not available
UsePreprocessorOptions	false	Not available

Option	Default	Corresponding Preference
PreprocessorOptions	""	Not available
PtrPrefix	ptr_	Not available
KeywordPrefix	keyword_	Not available
ArrPrefix	arr_	Not available
IncompletePrefix	incomplete_	Not available
TplPrefix	tpl_	Not available
UscoreSuffix	uscore	Not available
NoAbsolutePath	false	Not available
GenerateCPPTypes	false	Not available
OptimizeClassPointers	false	Not available

MAKEOPTIONS Section

Option	Default	Corresponding Preference
Capitalization	AsDefined	*Organizer*Capitalisation*
CCompilerDriver	false	Not available
CompileAndLink	true	*Organizer*CompileAndLink*
FileNamePrefix	""	*Organizer*FileNamePrefix*
GenerateASN1Coder	false	*Organizer*GenerateASN1Coder*
GenerateCode	true	*Organizer*GenerateCode*
GenerateEnvFunctions	false	*Organizer*GenerateEnvFunctions*
GenerateEnvHeader	false	*Organizer*GenerateEnvHeader*
GenerateSDLCoder	false	*Organizer*GenerateSDLCoder*
GenerateSignalNumbers	false	*Organizer*GenerateSignalNumbers*
Kernel	"SCTADEBCOM"	*Organizer*Kernel*
MakefileMode	Generate	*Organizer*MakefileMode*
PrefixType	Full	*Organizer*PrefixType*

Option	Default	Corresponding Preference
Separation	No	*Organizer*Separation*
StandardKernel	true	*Organizer*StandardKernel*
TargetLanguage	Cbasic	*Organizer*TargetLanguage*
UserKernel	""	*Organizer*UserKernel*
UserMakefile	""	*Organizer*UserMakefile*
UserTemplate	""	*Organizer*UserTemplate*
XCodeGenerator	X	*Organizer*XCodeGenerator*

PRINTOPTIONS Section

Option	Default	Corresponding Preference
BackwardReferences	true	*Print*BackwardReferences*
BlackAndWhite	false	*Print*BlackAndWhite*
DestinationFormat	PSFile	*Print*DestinationFormat*
FirstPageNo	1	Not available
FooterFile	""	*Print*FooterFile*
ForwardReferences	true	*Print*ForwardReferences*
FrameMakerCommand	"imaker"	*Print*FrameMakerCommand*
HeaderFile	""	*Print*HeaderFile*
MarginLeft	100	*Print*MarginLeft*
MarginLower	250	*Print*MarginLower*
MarginRight	100	*Print*MarginRight*
MarginUpper	420	*Print*MarginUpper*
OnlyChaptersInTOC	false	*Print*OnlyChaptersInTOC*
OrganizerView	true	*Print*OrganizerView*
PageMarkers	false	*Print*PageMarkers*
PaperFormat	A4	*Print*PaperFormat*

Option	Default	Corresponding Preference
PrintChapter	true	*Print*PrintChapter*
PrintCollapsed	true	*Print*PrintCollapsed*
PrinterCommand	"lpr -h -r"	*Print*PrinterCommand*
PrinterFile	""	Not available
PrintFrom		Not available
PrintTo		Not available
PrintToFile	true	Not available
TableOfContents	false	*Print*TableOfContents*
WordImageFormat	Normal-Dot	*Print*Word*ImageFormat*
WordUserDefined-Height	2470	*Print*Word*UserDefined-Height*
WordUserDefined-Width	2470	*Print*Word*UserDefined-Width*

System Window State File

When the system file is saved, a second file is saved as well; the system window state file. This file contains information about window positions and sizes. If an editor window is not open when the system file is saved, the last known position and size of that window is saved instead. While the system file uses the `.sdt` extension, the system window state file uses the `.sdt.state` extension.

If the Organizer finds a system window state file when opening a system file, the positions and sizes of the Organizer and editor windows are restored to the positions they had when the system file was saved. Note that the editor windows are not restored until they are opened from the Organizer.

Control Unit File

In addition to the System file (see "System File" on page 189), the Organizer also manages *Control Unit* files, `.scu` files. These control units are introduced to facilitate the workgroup (multiuser) support when working with an SDL system managed by the SDL Suite, and merging the individual results to a common system file.

General

The multiuser support is based on, from a revision control point of view, letting the user split the system file into several files. Information that is updated often should be split and stored in control unit files.

When and where to split the SDL structure is an active action taken by the user.

The multiuser mode is effective when control unit files are used. The use of the control unit files is optional, whereas the system file is mandatory. The Organizer thus always requires a system file and can manage multiple control unit files. By not taking advantage of control units simply means using the Organizer and system files as was done before, i.e. in previous versions (3.0X/3.1X/3.2).

Definitions

A definition for a *system file* (`.sdt` file) from a revision control point of view: a file that contains structure and state information for a document system as seen by one user. This file is normally not considered as an essential part of the system and is not suitable for revision control as it is private to one user.

- The diagram structure information is common to all developers of the system in a multiuser environment.
- The state information is specific for one user.

A similar definition for a *control unit* file (`.scu` file): a file that contains structure information for a subset of a document system and which is common to all users that work with the document system. Control unit files are part of the document system and are suitable for revision control.

The control unit files can be inserted and made visible in the Organizer's file structure view. The control unit files contain information that is specific to the Organizer file structure for diagrams, modules and chapters.

Splitting the System File

The users control what parts of the file structure managed by the Organizer need a separate control unit file. When several users work on a system, the management of the system file may be difficult to synchronize. This problem is solved if every user or group have control over their own part of the system file.

The Organizer structure information is hence split into several files – control unit files. The decision of where and how many control unit files are needed is left to the users; the idea being to partition the diagram system according to the user's work responsibilities and assign control unit files to these partitions. An example can be a large SDL diagram system where there are several blocks on the system level developed in parallel by different developers. Each block could then be associated with a control unit file. Now, the blocks can be updated independently and the changes are shaping the local control unit files. There is no need to manually merge changes into the system file when the work is done – the management of control unit files performed by the Organizer also includes the merge.

The user decides how control unit files and the system file cooperate to produce the Organizer view of the document system. Two scenarios are possible here. In one scenario the user decides that the system file is valid and the diagram structure is read from it. This is done when the user opens the system file and loads the document system into the Organizer. In the other scenario the control unit files are used to load the Organizer view and automatically update it accordingly to a specific revision for the document system – the system file is used only to fill in document state information (if any is available). This is done with a menu command when control unit files are used to explicitly update the document system or some part of it.

An Example

To illustrate the use and contents of control unit files and facilitate the understanding, let us discuss the topic with a simple example as input.

Example 5 ----------

Say we have the following system managed by the Organizer:

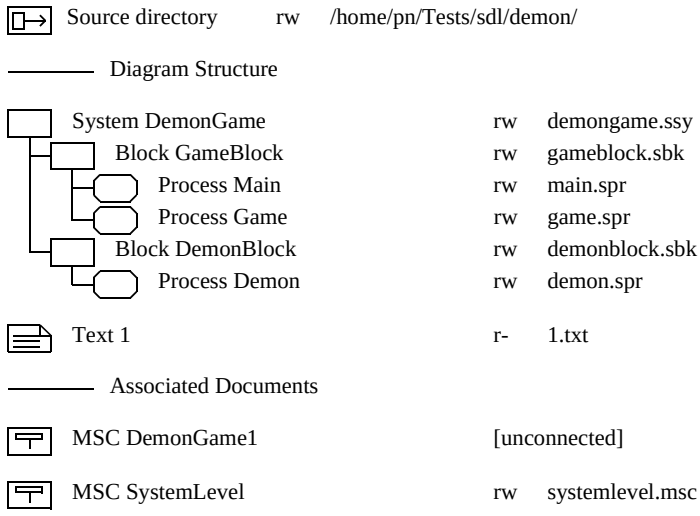


Figure 55: A simple system, managed by the Organizer

Let us say that we create control units (using the *Configuration > Group File* command) for the following items:

- One control unit that holds the entire system file together
- One control unit for the whole diagram structure
- One control unit for the system DemonGame
- One control unit for the block GameBlock.

We would now have a system that looks like this:

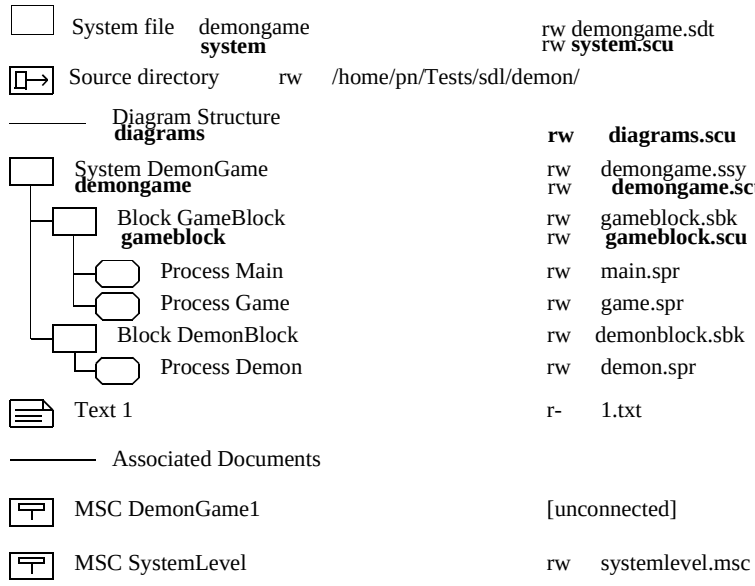


Figure 56: The same system, now with control units

The control units are depicted in bold face.

The [DIAGRAMS] section of the `demongame.sdt` file will contain:

```
[DIAGRAMS]
-2 102 system system.scu 0 0 - ""
-1 102 diagrams diagrams.scu 0 0 - ""
-1 23 "Diagram Structure" %unconnected% 0
0 102 demongame demongame.scu 0 0 - ""
0 1 DemonGame demongame.ssy 0 0 demongame ""
1 102 gameblock gameblock.scu 0 0 - ""
1 2 GameBlock gameblock.sbk 0 0 gameblock ""
2 5 Main main.spr 0 0 main ""
2 5 Game game.spr 0 0 game ""
1 2 DemonBlock demonblock.sbk 0 0 demonblock ""
2 5 Demon demon.spr 0 0 demon ""
0 33 1 1.txt 0 0 - ""
-1 23 "Associated Documents" %unconnected% 0
0 16 DemonGame1 %unconnected% 0 0 - ""
0 16 SystemLevel systemlevel.msc 0 0 - ""
```

The files `system.scu, diagrams.scu, demongame.scu` and `gameblock.scu` will contain the following information:

```
system.scu:
-1 102 diagrams diagrams.scu 0 0 - ""
-1 23 "Associated Documents" %unconnected% 0
0 16 DemonGame1 %unconnected% 0 0 - ""
0 16 SystemLevel systemlevel.msc 0 0 - ""

diagrams.scu:
-1 23 "Diagram Structure" %unconnected% 0
0 102 demongame demongame.scu 0 0 - ""
0 33 1 1.txt 0 0 - ""

demongame.scu:
0 1 DemonGame demongame.ssy 0 0 demongame ""
1 102 gameblock gameblock.scu 0 0 - ""
1 2 DemonBlock demonblock.sbk 0 0 demonblock ""
2 5 Demon demon.spr 0 0 demon ""

gameblock.scu:
0 2 GameBlock gameblock.sbk 0 0 gameblock ""
1 5 Main main.spr 0 0 main ""
1 5 Game game.spr 0 0 game ""
```

Format and Structure of the Control Unit File

Format

The format of the control unit file complies with the syntax of the [DIAGRAMS] section in the system file (see "Format of the System File" on page 190). There are however some exceptions:

The first line of the scu file indicates the source directory to use for relative files mentioned in the scu file. This information is saved in the same way as the same information is saved in the system file. By having this information in the scu file, it is possible to re-use/reference the same scu file from different system files with different source directories.

`<viewState>`
: This field is only used for the separation value - the value is therefore either 0 or 1.

`<timeStamp>`
: This field is not used - the value is always 0.

`<associations> ::= " <qualifiernameList> "`

A list of quoted qualifier names, each name denoting a document. See Example 6, below.

```
<dependencies> ::= " <qualifiernameList> "
```

A list of quoted qualifier names, each name denoting a document.

```
<qualifiernameList> ::= $ | (<qualifierName>
                          (, ($ | \NL) <qualifierName>)*)

<qualifierName> ::= ' <qualifierTuple>
                    (/<qualifierTuple>)* '

<qualifierTuple> ::= <integer> <string>
```

<string> is the name of the system file, chapter, module or diagram.

Example 6 ——————————————————————————

```
" '23 Diagrams Area/1 DemonGame/2 DemonBlock', \
'23 Diagrams Area/1 DemonGame/2 GameBlock' "
```

————————————————————————————————

Structure

Associations between Control Units and Document Structures

A control unit file is associated with a document structure in the Organizer. The control unit file holds information about the top node of the structure and information about lower level documents under the top node and/or other lower level control unit files.

Levels (Indents)

Document nodes in a control unit file have a relative level (i.e. `<indent>`) starting from 0. (Exceptions: a chapter node and an control unit file node associated with a chapter have always level -1). This gives the flexibility that the control unit file can be associated with a node on different levels in the system structure of documents. To get the actual level for a document node from an control unit, the control unit file node level should be added to the document's relative node level.

Example 7: For the control unit level 1 from Example 5 on page 202

```
0 2 GameBlock gameblock.sbk 0 0 gameblock ""
1 5 Main main.spr 0 0 main ""
1 5 Game game.spr 0 0 game ""
```

————————————————————————————————

Hierarchy

The control unit files are hierarchical. The top control unit file is for the whole system of documents in the Organizer. We give this control unit file level (-2). If this file has an information line in the system file this means that the top of the whole system structure is found in that file. Other structure information about diagrams mentioned in the system file is then only a mirror of the structure information found in the control unit hierarchy.

Example 8:

```
-2 102 system system.scu 0 0 - ""
```

Modularity

The user can choose to use a control unit file only for a specific chapter, module or diagram structure (without a top level control unit system file). This means that not all information in the system file is a mirror for structure information from control unit files. Some of the information can be actual structure information that is not controlled by control unit files. When the Organizer expands structure information from control unit files it tries to match that information against the contents in the system file. This matching is necessary because only the system file holds state information about documents. The structure information from control unit files takes precedence over structure information found in the system file. If there is an inconsistency between the control unit file structure information and the system file, the control unit information takes precedence and the SDL Suite contents are treated as an error situation. Say, for instance, that according to a control unit file there are three blocks in an SDL system but there are four blocks in the system file. The extra block is probably old information.

Example 9: A control unit file for a chapter

```
-1 102 diagrams diagrams.scu 0 0 - ""
```

Order of Appearance

A control unit information line in the system file is followed by a mirror information line of the top node that the control unit file is associated with. If the Organizer cannot open the control unit hierarchy the mirror information will function as a back-up document structure information.

Example 10 ----------

```
-1 102 diagrams diagrams.scu 0 0 - ""
-1 23 "Diagram Structure" %unconnected% 0
```

Batch Facilities

SDL Suite supports operations in batch mode, such as *printing* the documentation, *analyzing* an SDL structure and *generating code*. These operations are managed by the Organizer in a batch, windowless mode.

Batch Syntax

The syntax for a batch command is:

```
sdtbatch <systemfile> <options>
```

Where `<systemfile>` and `<options>` are described in "Batch Options" on page 212.

When the `sdtbatch` command is issued, the Organizer interprets the command line arguments, performs the requested operation and then terminates. Error logging is performed by the Organizer.

Information concerning the batch job is directed the standard output device.

> **Note:**
>
> When executing a Batch command, external service management via the PostMaster interface is disabled.

Print Multiple Files

Syntax:

```
sdtbatch -p[rint] systemfile \
         [-s[elect] documentfile] \
         [-o[ption] optionfile]
```

Prints the contents in the system file. Corresponds to clicking the *Print* quick button.

Print One File

Syntax:

```
sdtbatch -p[rint] \
         -s[elect] documentfile \
         [-o[ption] optionfile]
```

Prints one file.

Compare Two Diagram Files

> **Note:**
>
> This sdtbatch command produces a textual difference report. To get a graphical difference report, consider using the -grdiff option with the sdt startup command. Read more about this in chapter 2, *Introduction to the SDL Suite*.

Syntax:

```
sdtbatch -d[iff] [ -b ] [ -n ] \
          documentfile1 documentfile2
```

Compares two SDL, HMSC or MSC diagram files. By default, a summary of how many differences that were found is printed. When using the `-b` option (verbose), information about every differing object is presented, including an SDT reference to the object in question. The use of the `-n` option corresponds to setting the option Ignore moved or resized objects to off. Normally this means that moved and resized symbols will not be detected as being different.

Analyze

Syntax:

```
sdtbatch -a[nalyse] systemfile \
        [-s[elect] diagramfile] \
        [-o[ption] optionfile]
```

Analyzes the contents in the system file. Corresponds to the *Analyze* quick button.

Make

Syntax:

```
sdtbatch -m[ake] systemfile \
         [-s[elect] diagramfile] \
         [-o[ption] optionfile]
```

Makes a target as specified by the system file. Corresponds to the *Make* quick button.

Extract Text

Syntax:

```
sdtbatch -extract [ -comment ] \
```

```
[ -keyword <keyword> ] <diagram file>
```

Extracts text from the diagram file. As default, all text in the diagram is shown. By using the *-comment* option, only text in comment symbols and /* C-style comments */ in other symbols is shown. The *-keyword* option limits the shown text to text from symbols containing the <keyword>. The <keyword> itself is not shown. For instance, if you have a comment symbol with the following text:

```
diagramversion 1.44
```

Then...

```
sdtbatch -extract -comment \
         -keyword diagramversion a.ssy
```

...will produce the following output:

```
1.44
```

Show License Information

Syntax:

```
sdtbatch -licenseinfo
```

Show information about license holders and number of available licenses, as textual output in the console window where the command was entered.

Show Version Information

Syntax:

```
sdtbatch -v[ersion]
```

Reports the current program version number for the installed tools.

List Files

Syntax:

```
sdtbatch -files systemfile
```

Lists all files referenced from the system file, as textual output in the console window where the command was entered.

Pack Files

Syntax:

```
sdtbatch -pack systemfile [ archivefile ]
```

Create an archive file (*.tgz) for a system, using default pack settings. If the archive file name is not given, then an archive file will be created in target directory, with a default name.

Unpack Files

Syntax:

```
sdtbatch archivefile [ directory ]
sdt archivefile [ directory ]
```

Extracts an archive file into the current or supplied directory. If executed in normal mode it will open the systemfile.

Change Case

Syntax:

```
sdtbatch -changecase cross-reference-file
```

Change the case for all words found in all referenced files in the cross reference file to the case used in the declaration.

The command will also change the case for all keywords. These should be located in a special keyword file found in the same directory, with the same name as the name of the given cross reference file but with the extension `.key`.

The command does only work with the standard SDL text editor.

To generate a more extensive cross reference file and a file containing keywords, run the `sdtbatch -analyze` command with the options

```
SDLKeywordFile=True

SetPredefinedXRef=True
```

added to the option file. The analyzer will then generate more information in the cross reference file, case-sensitive `.xrf`, and furthermore a keyword file, case-sensitive.key, in the target directory of the system.

Configuration Update

Syntax:

```
sdtbatch -r[ecursive] systemfile \
         [-s[elect] diagramfile] [-u[pdate]]
```

Does a recursive update of the structure in the Organizer according to the Control Unit files in the system file. Corresponds to the *Configuration > Update* menu choice having the diagram file selected. Without the -s option the action is as if the systemfile has been selected.

Checking diagrams for duplicated object IDs

Syntax:

```
sdtbatch -checkdiagrams systemfile
sdtbatch -checkandfixdiagrams systemfile
```

Normally each object in an SDL diagram have a unique ID within the diagram. This ID can be shown for example when using the SDL editor command Show GR Reference. For some existing SDL diagrams, especially diagrams that have been created using the Merge or Add Differences functions, the same ID has been used by more than one object. This will result in unpredictable behaviour when using GR References, e.g. when transforming the system to case-sensitive SDL or when an analysis error should be shown, the wrong symbol might appear. Also if the two objects having the same ID are on the same SDL Page, the lines can be connected to symbols in a way that is different from the way you expect by looking at the geometrical layout of the lines. This can give strange errors when generating SDL/PR, giving strange analysis errors. This problem has been fixed in versions 4.2.7, 4.3.3, and 4.4, therefore diagrams that are saved with these or later versions will not have this problem.

To test the diagrams for duplicated IDs you can run the batch command `-checkdiagrams`. The command will produce a report on duplicates for all the SDL diagrams.

Whenever there are problems with duplicates you can run `-checkandfixdiagrams`. This will fix all duplicates and save the diagrams such that all diagrams are free from duplicated IDs.

Batch Options

- `<systemfile>`

  `<systemfile>` is an Organizer *System File*. A valid system file should be supplied.

- `-select <file>`

`<file>`, which should be included in the document structure, (associated documents are also included if the `-print` option specified) becomes the selected object upon which the command is applied. This option corresponds to a manual selection of the document in the Organizer drawing area.

A full file path for the file must be supplied.

- `-o[ption] <optionfile>`

  `<optionfile>` is a file with the same syntax as a *System File*, containing options which supersede the options found in the system file. An optional section CPP2SDLOPTIONS can manually be entered into the option file to supersede the SDL systems cpp2sdl import specification files. An arbitrary number of options may be supplied, but options in the file must be preceded with an option section field as found in the system file.

Example 11: CPP2SDL section in a batch ".sdt" -file ----------

```
[CPP2SDLOPTIONS]
InputLanguageC=False
InputLanguageCpp=True
InputLanguageMicrosoft=False
InputLanguageGNU=False
RTTI=False
RecognizeSDLsorts=False
ObjectSlicing=False
Preprocessor=""
UsePreprocessorOptions=False
PreprocessorOptions=""
PtrPrefix=ptr_
KeywordPrefix=keyword_
ArrPrefix=arr_
IncompletePrefix=incomplete_
TplPrefix=tpl_
UscoreSuffix=uscore
NoAbsolutePath=True
GenerateCPPTypes=False
OptimizeClassPointers=False
```

An optional section PRINTSETUPOPTIONS can manually be entered into the option file to identify the print setup options to be used during batch printing.

Example 12: PRINTSETUP section in an options file used for batch printing

```
[PRINTSETUPOPTIONS]
TOCAfterText=True
OnlyChaptersInTOC=False
OrganizerScale=FitPage
LinkScale=100
OMScale=100
HMSCScale=100
SCScale=100
DPScale=100
SDLInteractionScale=FitPage
SDLFlowchartScale=FitPage
SDLOverviewScale=100
MSCScale=FitWidth
CoverageScale=100
IndexScale=100
TypeScale=100
MSCInstanceRuler=True
OrgViewPageRef=True
```

Note:

Note that if the options contains a filename in quotes all backslashes must be escaped with one more backslash. Example of correct file-name uses:

`PrinterFile=d:\tmp\printfile.ps`

or

`PrinterFile="d:\\tmp\\printfile.ps"`

- `-u[pdate]`

  If this option is supplied, the system file will be saved after the batch operation has finished.

Windows and UNIX File Compatibility

Systems can be created with SDL Suite and TTCN Suite on both the UNIX and Windows platforms. To allow systems to be accessed on both platforms, SDL Suite and TTCN Suite provides a file compatibility concept.

The *drive table* contains a mapping between Windows and UNIX directory paths. This mapping is included in the system file to allow file specifications in the system file to be translated to the correct format on both platforms. See "The Drives Section" on page 192 for more information.

The menu choice *PC Drives* is available in the Organizer's *File* menu, and allows editing the drive table that is stored in the system file. See "PC Drives" on page 73 for more information.

The drive table takes care of translating absolute paths between the two different file name formats (UNIX and Windows). On a UNIX system, some restrictions apply to file names to be compatible with Windows. These restrictions are checked whenever the user changes a file specification that is stored in the system file; if they are not followed, an error dialog is shown and the user is returned to the dialog where the file was specified. See "Filename Error Dialogs (UNIX only)" on page 33 for more information.

File specifications are always stored in the correct format according to the current platform.

In general, diagrams with file specifications that are of incorrect format or that cannot be translated to the correct format, are marked as Invalid in the Organizer.

Chapter

3 *The Preference Manager*

The Preference Manager is used for setting up the default behavior of the SDL Suite tools, and the common tools.

This chapter is a reference to the Preference Manager; the functionality it provides, its menus, windows and symbols and a description of the preference parameters.

For a guide to how to use the Preference Manager to customize the behavior, see chapter 4, *Managing Preferences*.

Preferences Manager User Interface

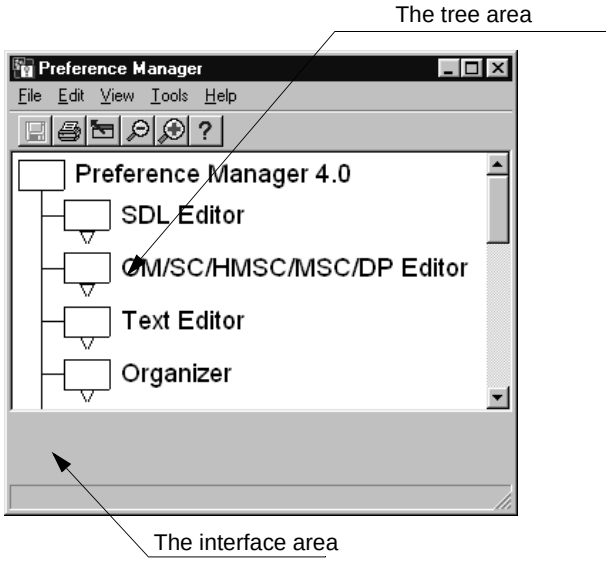


Figure 57: The Preference Manager window

The items that build up the Preferences Manager window are:

- The *Tree Area*
- The *The Interface Area*
- The *Menu Bar*

Tree Area

The *tree area* displays the tools that obey to the preference mechanism in the tools and their respective preference parameters. The notation uses a graphical approach featuring a tree with icons symbolizing the various items that are managed by the tool.

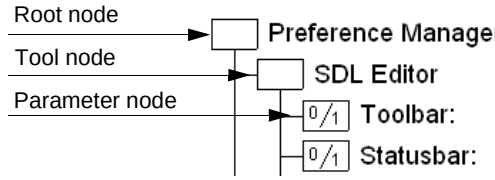


Figure 58: Node in the tree area

Node Syntax

Each node in the tree represents either a:

- *root node*

  The root node is identified by the name Preference Manager, followed by a version number.

- *tool node*

  A tool node is identified by the name of a tool.

> **Note:**
> The Preference Manager only supports the SDL Suite tools, and the common tools. See chapter 29, *Customizing the TTCN Suite (on UNIX)*, for information about how to customize the TTCN Suite **on UNIX**.

- *parameter node*

  Each parameter node in the tree is identified with an icon showing:

  – the *name* of the parameter

  – the *current value* for the parameter

  – the *saved value* and the *default value* within one or two parentheses: `(( ))`

  – the *Preference Source* within brackets, `[]`, i.e. whether it is a *company* or *project* or *user-defined* setting

  – a *descriptive text* providing additional information about the preference parameter.

Example 13: A Line in the Preferences Window ––––––––––––––––––

Given the following line in the Preferences window, with all *View Options* turned “on”:

Figure 59: A line in the Preferences window

The information on the line in Figure 59 should be interpreted as:

Item	Interpretation
	The *icon* informing that the preference parameter is a boolean parameter (0/1, false/true or off/on parameter.
`Editor*SymbolMenu`	The *name* of the preference parameter, consisting of • the name of the tool it affects, `Editor` • the name of the parameter, `SymbolMenu` Separated with an asterisk
`off`	The *current value* for the parameter.
`(off)`	The *saved value* for the parameter.
`((on))`	The *default value* for the parameter.
`[User]`	The parameter’s *source*, in this case the parameter is defined by the user.
`Show the symbol menu`	The parameter’s *description*, i.e. what tool property the parameter affects.

––

The tree is a visualization of the merged preference parameters, supplied from different sources (see “Search Order When Reading Preference Parameters” on page 238), as the tools will perceive them. Furthermore, each node in the tree can be collapsed or expanded (see “Collapse” on page 229) in order to reduce or extend the amount of information that is visible, thus facilitating the work with the tool. It is

also possible to filter the information that is displayed with respect to various view options (see "View Options" on page 230).

Dirty Notification

When a preference parameter is modified but not saved, this is indicated by the icon appearance changing from a light color to a gray pattern. The parent *tool node* and the *root node* are also marked as dirty when any parameter is changed.

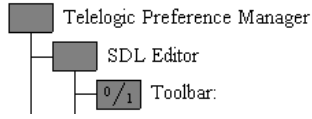


Figure 60: Nodes marked as dirty

The Interface Area

The interface section located under the tree view is where values are edited in an *Edit Area*. The type of *Edit Area* varies with the type of value selected.

There are different types of preference parameters:

- *Textual Preference*
- *Option Menu Preference*
- *Option Menu Textual Preference*
- *Font Preference (Windows only)*
- *Boolean Preference*
- *Ranged Integer Preference*
- *File Preference*
- *Directory Preference*
- *Color Menu Preference*

Icon	Edit Area
T	.bld Textual Preference – The preference parameter is edited as a text string.

Icon	Edit Area
{x,y}	 Option Menu Preference – The preference parameter may be selected from a number of predefined values in the option menu.[a]
E	**On UNIX:**  **In Windows:**  Option Menu Textual Preference – The preference parameter may be selected from a number of predefined values in the option menu.[a] An alternative is to type in the value into the text field, which grants access to not predefined values.

Icon	Edit Area
F	 Font Preference **(Windows only)** – The font must be selected by clicking the folder button, which opens a Microsoft Windows Font dialog. It is not possible to edit the text field to the left of the folder button. – In the dialog, all fonts in the system are listed. It is also possible to select font style and size, but these values have no effect; only the specified font name is used. – Any font name can be entered by editing the *Font* text field in the font dialog. This possibility is useful when setting a font for printing that is not available in the system, but exists on the printer. – The three font names “Times”, “Helvetica” and “Courier” are recognized. These are translated into the TrueType fonts Times New Roman, Arial and Courier New, respectively. – When using other fonts than the three mentioned above for printing, the preference Print*Destination-Format should be set to *MSWPrint* to get the best result for symbols that are adjusted to the text size.
0/1	 Boolean Preference – The value is changed by clicking the corresponding radio button.
[a..b]	 Ranged Integer Preference – The preference parameter may be assigned any integer value within the range supported by the slide bar.

Icon	Edit Area
	 File Preference – The preference parameter may be assigned any file name, entered as a text string into the text field or selected in a dialog issued by the folder button. Directory Preference – The preference parameter may be assigned any directory name, entered as a text string into the text field or selected in a dialog issued by the folder button.
	 Color Menu Preference – The preference parameter may be selected from a number of predefined values in the option menu.

a. When the mouse pointer is moved over a value in the option menu, a short description of the value is printed in the status bar. Values whose description starts with “(UNIX only)” or “(Windows only)” should only be selected on a UNIX system or a Windows system, respectively.

Selecting a parameter and editing its value with the provided UI device changes its value. The parameter’s icon is set as dirty (see “Dirty Notification” on page 221).

Menu Bar

The menu bar contains the following menus:

- *File Menu*
- *Edit Menu*
- *View Menu*
- *Tools Menu*
- *Help Menu*

File Menu

The *File* menu contains the following menu choices:

- *Save*
- *Revert*
- *Print*
  (see "The Print Dialogs in the SDL Suite and in the Organizer" on page 316 in chapter 5, *Printing Documents and Diagrams*)
- *Info*
- *Exit*
  (see "Exit" on page 15 in chapter 1, *User Interface and Basic Operations*)

Save

This menu choice saves the parameters that have been modified in the *User's Preference File*, **on UNIX** in `$HOME/.sdtpref`, and **in Windows** in `%HOME%\sdt.ini`

But, before saving the preference settings, the user will be warned if there are modified preferences that are defined as company or project preferences:

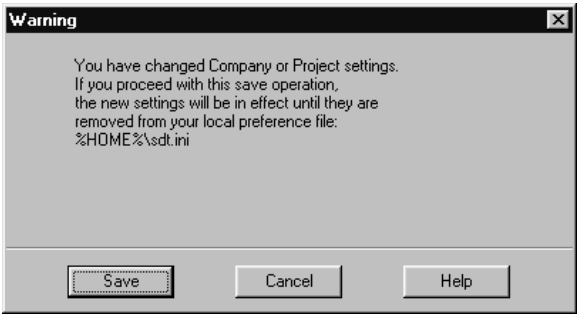


Figure 61: Attempting to save a modified project or organization parameter

- Clicking *Save* saves **all** modified preference parameters on the *User's Preference File*, which may implicate a violation of the rules set up in the project!

Caution!

Consider the implications before you click *Save*.

When the save has been performed the user will be warned that changes will not be in effect before the tools have been restarted.

Revert

The menu choice reverts all preference parameters to the currently *saved values*, as they were defined when starting the Preference Manager. The *Revert* command requires to be confirmed, in case any preference parameters are modified and have not been saved:

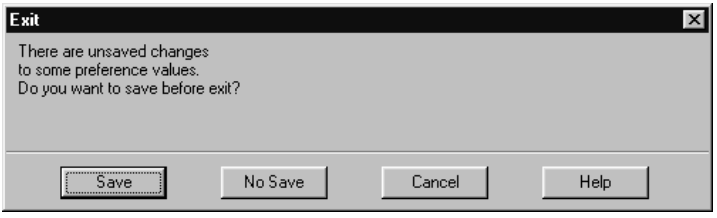


Figure 62: The Confirm to Revert dialog

Info

This menu choice issues a message with information about the preference source for the preference parameters.

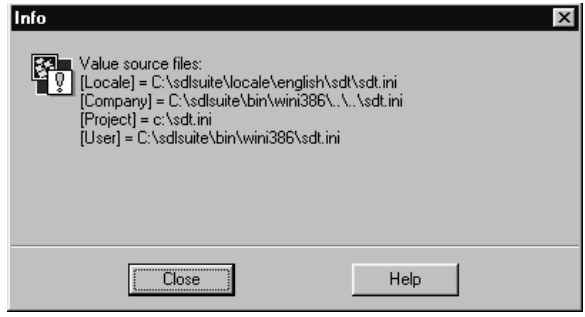


Figure 63: Information about preference sources

The message shows four items:

- `[Locale] - <file specifier>`
  The location of the locale preference file
- `[Company] - <file specifier>`
  The location of the company / organization preference file
- `[Project] - <file specifier>`
  The location of the project preference file
- `[User] - <file specifier>`
  The location of the user's preference file.

See "Preference Files" on page 235 for more information on preference files and search order.

Edit Menu

The *Edit* menu contains the following menu choices:

- *Set Saved Value*
- *Set Default Value*
- *Unsave*

Set Saved Value

This command restores the selected parameter to its currently *saved* value. The menu choice is also available from the pop-up menu.

Set Default Value

This command restores the selected parameter to its *default* value (i.e. factory setting). The menu choice is also available from the pop-up menu.

Unsave

This command allows to remove a preference parameter from the *User's Preference File* when saving the file, i.e. *Unsave* it. The command is available on user-defined parameters only; user-defined parameters are shown with their *Value Sources* set to [User].

A dialog is displayed following invocation of the command:

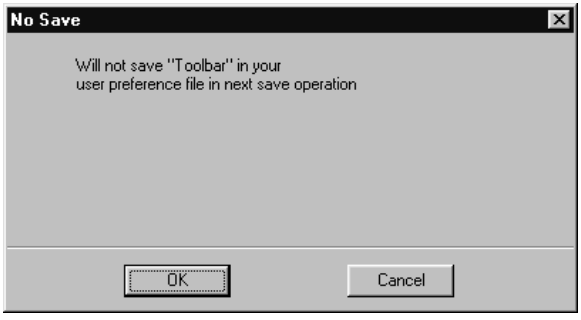


Figure 64: The Unsave dialog

- Clicking *OK*, the parameter's source changes to [User (No Save)]. The preference parameter will be removed from the user's prefer-

ence **after the next** *Save* command, changing its source to [Project] or [Company].

View Menu

The *View* menu contains the following menu choices:

- *Expand*
- *Expand All*
- *Collapse*
- *Collapse All*
- *View Options*
- *Set Scale*

Expand

This menu choice is available only if the currently selected object is a *tool node*. The command expands the subtree starting from the selected tool, making all parameters for that tool visible.

The operation is also available from the pop-up menu and with a double-click.

Expand All

This command expands the entire tree, thus making all parameter nodes visible.

The operation is also available from the pop-up menu.

Collapse

This menu choice is available only if the currently selected object is a *tool node*. The command collapses the parameter nodes belonging to the current tool, leaving only the tool node visible in the subtree.

The operation is also available from the pop-up menu and with a double-click.

Collapse All

This menu choice collapses all *tool node*s and hides the *parameter node*s. Only the *root node* and the *tool node*s are left visible.

The operation is also available from the pop-up menu

View Options

This menu choice issues a modeless dialog, the View Options dialog:

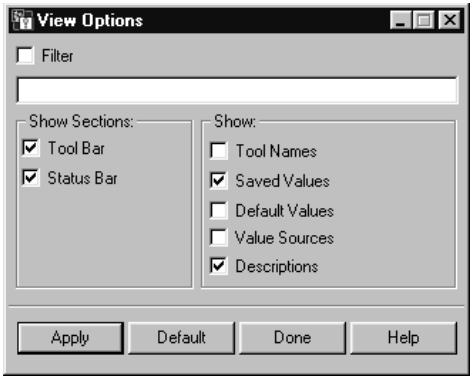


Figure 65: The View Options dialog

The dialog contains the following toggle buttons:

- A *Tool Bar* button
- A *Status Bar* button
- A *Filter Field and Filter Button*
- A *Tool Names* button
- A *Saved Values* button
- A *Default Values* button
- A *Value Sources* button
- A *Descriptions* button

Tool Bar

This button controls the presence of the quick buttons.

The tool bar is visible by default.

Status Bar

This option determines whether the *Status Bar* should be visible or not.

The status bar is by default visible.

Filter Field and Filter Button

If the *Filter Button* is on, the *Tree Area* will only show the preference parameters which name contain the text string that is entered in the *Filter Field*.

> **Note:**
>
> The name of a preference parameter consists of a combination of a tool name and of the parameter itself (see "Syntax of Preference Files" on page 237).
>
> The filter option compares the text strings without respect to upper or lower case characters.

By default, no filtering is applied.

Tool Names

This option controls whether the prefix in preference parameters (consisting of the tool name and of an asterisk) should be displayed or not (see "Syntax of Preference Files" on page 237).

Tool names are not displayed by default.

Saved Values

This option determines whether the *saved values* should be made visible or not. The saved values are the values that are currently stored on the preference files.

Saved values are displayed within one level of parentheses *(saved value)*. See Example 13 on page 220.

Saved values are displayed by default.

Default Values

This option determines whether the *default values* (i.e. factory settings) should be made visible or not.

Default values are displayed within two levels of parentheses *((default value))*. See Example 13 on page 220.

Default values are not displayed by default.

Value Sources

This option determines whether the *Preference Source* for the parameters should be displayed or not. A preference parameter may be retrieved from any of the following sources:

- The *company settings*
- The *project settings*
- The *user's own settings*

Source values are displayed within brackets [] and are not visible by default.

Descriptions

This option determines whether an informative text should be displayed or not. The text is purely informational and provides an additional description of the function of the preference parameter. See Example 13 on page 220.

Parameter descriptions are not displayed by default.

Set Scale

This menu choice opens a dialog where you may set the scale used in the preferences window. The scale may be set between 20% and 800%. 100% is the default.

Tools Menu

The *Tools* menu contains one menu choice:

- *Show Organizer*
  (see "Show Organizer" on page 15 in chapter 1, *User Interface and Basic Operations*)

Popup Menu

The contents of the popup menu depends on which node you click on.

On the Root Node

Collapse All	See “Collapse All” on page 229.
Expand All	See “Expand All” on page 229.

On a Tool Node

Collapse	See “Collapse” on page 229.
Expand	See “Expand” on page 229.

On a Value Node

Set Saved Value	See “Set Saved Value” on page 228.
Set Default Value	See “Set Default Value” on page 228.
Help	Request on-line help for the preference parameter in question.

Keyboard Accelerators

Apart from the general keyboard accelerators, as described in "Keyboard Accelerators" on page 35 in chapter 1, *User Interface and Basic Operations*, the following accelerators can be used in the References window:

Accelerator	Operation
`Arrow Up`	Move selection up among the expanded nodes
`Arrow Down`	Move selection down among the expanded nodes
`Page Up`	Scroll up
`Page Down`	Scroll down
`Home`	Scroll left
`End`	Scroll right
`Ctrl+Home`	Display first line
`Ctrl+End`	Display last line
`Space or Return`	Expand or collapse a node. Traverse through option menu. Toggle parameters on/off.
`F2`	Raise the pop-up menu.

Preference Files

All of the preference parameters along with their values (exception made for the default settings which are programmed into the tools) are stored on dedicated files, the *preference files*. The preference files may be edited with any text editor since the information is stored in textual form.

The Preference Manager manages these files and provides a graphical user support for customizing tools in an easy way.

> **Note:**
>
> The TTCN Suite cannot be customized by preferences. Instead, see chapter 29, *Customizing the TTCN Suite (on UNIX)*.

Preference Source

The following files build up the preference environment:

- The *Locale Preference File*
- The *User's Preference File*
- The *Project Preference File*
- The *Company Preference File*

The following items constitute the various sources for preference parameters. The source identifier may be made visible with the *Value Sources* options in the *View Options* dialog.

Source Identifier	Source Description
`[User]`	The *User's Preference File*
`[User (No Save)]`	The *User's Preference File*, will be removed from the user's preference file next time the parameters are saved
`[Project]`	The *Project Preference File*
`[Company]`	The *Company Preference File*
`[Locale]`	The *Locale Preference File*
`[Default]`	The default *Factory Settings*.

User's Preference File

This file stores the preferences parameters that a user has modified. The file is created / updated as a result of the *Save* command.

The user's preference file is identified **on UNIX** by:

```
$HOME/.sdtpref
```

and **in Windows** by:

```
%HOME%\sdt.ini
```

Note: $HOME and %HOME%

On UNIX, `$HOME` is assumed always to be defined in the computer environment.

In Windows, if `%HOME%` is not set, the user's preferences will be stored in the directory from which SDL Suite and TTCN Suite is started.

Project Preference File

This file stores the preference values that are to be applied to a specific instance of a project where the SDL Suite is used.

The project preference file is identified **on UNIX** by:

```
$SDTPREF/.sdtpref
```

and **in Windows** by:

```
%SDTPREF%\sdt.ini
```

Note: $SDTPREF and %SDTPREF%

`If $SDTPREF` **(on UNIX)** or `%SDTPREF%` **(in Windows)** is not set, the project preference file feature will be disabled. In a multi-user environment, it is strongly recommended to use a directory resident on a network drive for the project preferences. All project members should have read access to this directory.

Company Preference File

This file stores the preference values that are common for an entire company or organization and therefore should be applied to all projects and users of SDL Suite and TTCN Suite.

The company preference file is identified **on UNIX** by:

```
$telelogic/.sdtpref
```

and **in Windows** by:

```
<Installation directory>\sdt.ini
```

> **Note:**
>
> **On UNIX**, `$telelogic` is assumed always to be set in the environment.
>
> **In Windows**, to function properly (in the sense of being unique) in a multi-user environment, the company preference file should be stored on a directory on a network drive, e.g. if SDL Suite is installed on a network. Also, all users should have read access to the company preference file.

Locale Preference File

This file stores the preference values that are locale dependent and should therefore be applied to all projects and users of the SDL Suite tool.

Factory Settings

These are the settings that are defined by default in the SDL Suite and TTCN Suite tools. The factory settings will be used whenever a preference parameter could not be found in the *Preference Files*.

Syntax of Preference Files

A line in the preference file contains two items:

1. The first item specifies a tool and a parameter, separated by an asterisk.
2. The second item specifies the current value that is to be assigned to that parameter when starting up the tool.

The syntax is:

```
tool*parameter: value
```

Example 14: A Line in Preferences File ––––––––––––––––––––––

An example line in a preferences file could look like this:

```
Print*Scale:100
```
Meaning that the scale when printing is 100%.

The order of appearance of the parameters in a file is of no significance.

Search Order When Reading Preference Parameters

This section discusses how the tools read the preference parameters.

When a tool starts, it will automatically (i.e. no user-interaction is required) retrieve the preference values **in an incremental way**, following the order of search described below and illustrated in Figure 66 and Figure 67:

1. First, fallback values (default factory settings) are loaded, followed by the locale settings.
2. Then, if the environment variable `SDTPREF` is set:
   - If `$SDTPREF/.sdtpref` **(on UNIX)**, or `%SDTPREF%\sdt.ini` **(in Windows)** exists, that file is read **incrementally**[1] as the project preference file.
   - Otherwise, the tools will look for organization preferences in the installation directory, in the file designated by `$telelogic/.sdtpref` **(on UNIX)** or `<`*`Installation directory`*`>\sdt.ini` **(in Windows)**.
3. Finally, if the `HOME` environment variable is set, and `$HOME/.sdtpref` **(on UNIX)** or `%HOME%\sdt.ini` **(in Windows)** exists, that file is read **incrementally** as the user's preference file.
   - Otherwise, if `HOME` is not set, the preference file in the directory from which SDL Suite was started is read **incrementally** as the user's preference file.

The method above allows to configure a tool environment by defining some preferences as company-wide or project-wide, while allowing users to customize other preferences.

---

1. Incrementally in the sense that only parameters that are stored in the project preference file will override previous settings.

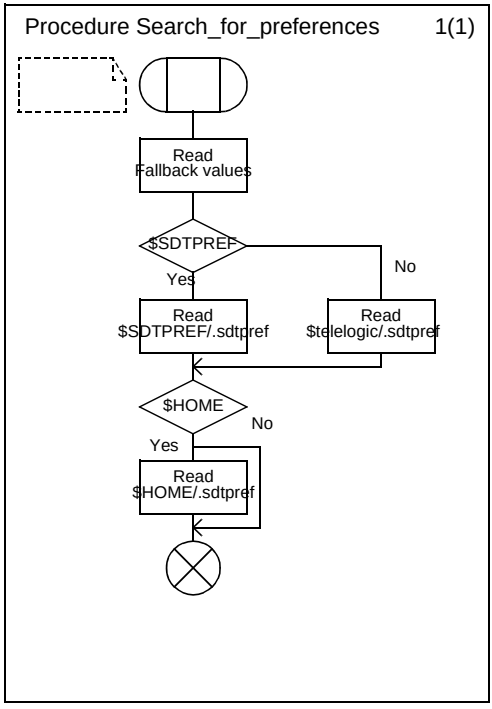


Figure 66:How the tool searches for preference parameters **(on UNIX)**

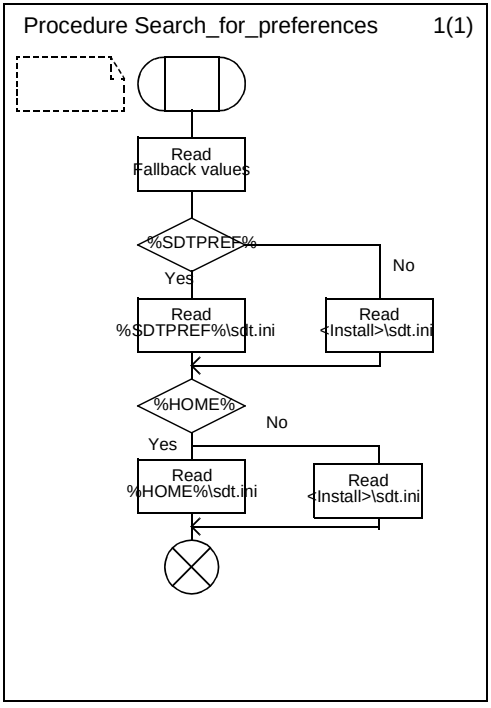


Figure 67: How the tool searches for preference parameters **(in Windows)**

Save of Preference Values

When a preference parameter is saved, it is always stored as a user-defined preference parameter in the user's preference file, i.e. `$HOME/.sdtpref` **(on UNIX)**, or `%HOME%\sdt.ini` **(in Windows)**, if `HOME` is set. If `HOME` is not set, the preference parameters are stored in a preference file in the directory from which SDL Suite was started.

The above implies that modifying a parameter that was originally defined as a project or organization parameter, the parameter will from now on be considered as a user-defined parameter. The user is however notified about this (see Figure 61 on page 226).

Preference Parameters

The following sections are a reference to the preference parameters in the tools with their default values and a description of what the parameter affects.

> **Note:**
>
> All preference parameters for the tools that are available on UNIX and in Windows are listed and described. This is to ensure compatible interchange of preferences between UNIX and Windows platforms, even if the preference parameter has no impact on one of the platforms. When this is the case, it is clearly noted.

For clarity reasons, the prefix that identifies the tool that the parameter applies to is specified once only, at the beginning of each tool section.

Preference parameters are grouped tool-wise. Within each tool section, parameters are listed in alphabetic order.

Where the parameter is specified with a numeric default value, the sort is specified within parentheses "()". Parameters whose value is specified as a text and which do not have any default value are identified as "". Possible values are not specified; the user should run the Preference Manager in order to obtain a reference to the permitted values.

The following preference groups exist:

The TTCN Suite tools cannot be customized by preferences. Instead, see chapter 29, *Customizing the TTCN Suite (on UNIX)*.

Common Preferences

These preferences apply to tools that are available in all SDL Suite and TTCN Suite tools.

Generic Preferences

Prefix: SDT*

Generic preferences may be used by tools that have not defined any specialization for these preferences.

Preference Parameter	Default	Affects
AllowSpaceInFileNames	off	Whether SDL Suite or TTCN Suite allows the user to add files with names containing spaces. The default setting of not allowing file names with spaces is set to ensure maximum compatibility with other tools that may be connected to SDL Suite and TTCN Suite. Unless you are using SDL Suite and TTCN Suite stand-alone, you should carefully consider if your environment can correctly support spaces in file names before enabling this feature.
CaseSensitive	off	Whether the SDL language should be case sensitive or not. It will effect the syntax check in the SDL Editor and the work of the Analyzer. The default setting is as SDL96, i.e. the language is not case sensitive. Also used as the default value for the Analyzer option *Case sensitive SDL* in the Organizer.

Preference Parameter	Default	Affects
CMIntegration	No integration	Specifies if SDL Suite and TTCN Suite are used together with an external configuration management tool. When set to *ClearCase*, the name of the view is shown in the Organizer window title.
Drives	""	The default contents of the The Drives Section in the system file for a newly created system.
EmacsCommand	"emacs"	The command used to start the Emacs text editor. **UNIX only**, see **Note: on page 241**.
FileNameCompletion	on	Whether file name completion is invoked by a completion character.
FileNameCompletionChar	""	The character that invokes file name completion.
FunctionFrequencyLogFile	""	File to save tool usage statistics on; see FunctionFrequencyLogging. In addition, tool usage statistics is also saved to a file `sdtfreq.log` in the installation directory.
FunctionFrequencyLogging	on	Whether to collect statistics on the usage of menu commands, quick buttons and accelerators. The tools that support collecting statistics are: • the Organizer • the SDL Editor • the OM Editor
ISpellCommand	"type %1 | ispell -w "_" -f %2 -a"	Template for the command used to communicate with ispell spelling checker.

Preference Parameter	Default	Affects
LicenseTimeout	0	The time interval (in hours) an interactive SDL Suite and TTCN Suite user is allowed to remain idle before licenses are automatically released. Setting this value to 0 disables the automatic release of licenses.
PrintFontFamily	“Times”	The font face to use when printing text.
RelativeSDTREF	off	If file names in SDT references should be expressed as relative to source directory or not.
Scale	100 (%)	The scale to use when displaying on screen.
ScreenFontFamily	“Helvetica”	The font face to use when displaying text on screen.
SearchUnderscoreAware	on	Decides if the search operation should find “a_ b” when searching for “ab” in SDL diagrams and MSCs.
Sound	on	The emission of an alert sound (beep) when performing an illegal operation.
StartInformationServer	on	If an Information Server should be allowed to start or not.
TextEditor	Telelogic	The text editor to use when editing text documents.
UseLowerCaseInFileNames	off	If turned on, saved diagram names are converted to lower case to provide UNIX/DOS file name compatibility. Activating this preference can lead to problems when accessing existing files with names containing non-lowercase letters.

Preference Parameter	Default	Affects
WordCommand	“Winword.exe”	The command used to start Microsoft Word. **Windows only**, see Note: on page 241.
TextWindowFontFamily	Courier New	The font used in text windows. **Windows only**, see Note: on page 241.
TextWindowFontHeigth	10	The size of the font used in text windows. **Windows only**, see Note: on page 241.
UserInterfaceFontFamily	MS Sans Serif	The font used in the user interface. **Windows only**, see Note: on page 241.
UserInterfaceFontHeight	9	The size of the font used in the user interface. **Windows only**, see Note: on page 241.
SimulatorExplorerFont-Family	Arial	The font used in the SDL Simulator and SDL Explorer user interface. **Windows only**, see Note: on page 241
SimulatorExplorer-FontHeight	11	The size of the font used in the SDL Simulator and SDL Explorer user interface. **Windows only**, see Note: on page 241.
TabSize	2	Number of spaces inserted when the TAB key is pressed in a text area. **Windows only**.

Organizer Preferences

Organizer Pack Preferences

Prefix: Organizer*Pack. For more information about the pack operation, see "Pack Archive" on page 66 in chapter 2, *The Organizer*.

Preference Parameter	Default	Affects
TarCommand	tar	The command used by the pack archive operation to access the tar utility. See "Additional required tools and utilities" on page 3 in chapter 1, *Platforms and Products*.
GzipCommand	gzip	The command used by the pack archive operation to access the gzip utility. See "Additional required tools and utilities" on page 3 in chapter 1, *Platforms and Products*.
ArchiveSdt	on	Decides if a system file adapted for the archive contents should be included.
OriginalSdt	off	Decides if the original system file should be included.
Packages	on	Decides if SDL package diagrams should be included.
OtherSDL	on	Decides if other SDL diagrams (system, block, process...) should be included.
MSC	on	Decides if MSC diagrams (MSC, HMSC) should be included.
UML	on	Decides if UML diagrams (object model, state chart, deployment) should be included.

Preference Parameter	Default	Affects
Chapter	on	Decides if chapters should be included.
Generic	off	Decides if files attached to the Organizer generic symbol should be included.
Tau	off	Decides if files attached to the Organizer Tau documents symbol should be included.
Rhapsody	off	Decides if files attached to the Organizer Rhapsody documents symbol should be included.
SimScript	off	Decides if simulator scripts (*.cui and *.com files attached to a plain text symbol) should be included.
OtherText	on	Decides if other text documents (not simulator scripts) should be included.
Scu	off	Decides if configuration unit files (*.scu) should be included.
Index	on	Decides if index files (*.xrf) should be included.
Header	on	Decides if header and footer files should be included.
Extensions	lst	Files in source or target directory with the extensions specified here (space separated) are included.

Other Organizer Preferences

Prefix: Organizer*

Preference Parameter	Default	Affects
AbsolutePath	off	If file names should be specified with an absolute directory path or not. See "Set Directories" on page 71 in chapter 2, *The Organizer*.
AllowImplicitTypeConv	off	The default value for the Analyzer option *Allow implicit type conversion*.
AllPreferences	off	If all preferences or if only the "diff" between current Organizer options and Organizer start-up options should be written as options in the system file. See "Options in the System File" on page 193 in chapter 2, *The Organizer*.
Areas	"Analysis Model,Used Files,SDL System Structure,TTCN Test Specification,Other Documents"	The names of the chapters in the Organizer's drawing area, separated by commas. The default is known as the *basic Organizer view*.
ASN1KeywordFileName	""	The default value for the Analyzer option *ASN.1 keyword substitution file*.
ASN1Suffix	".asn"	File name extension used to recognize ASN.1 files.
AutoCMUpdate	off	If this preference is on, then a configuration update operation will be done each time a system file is loaded in the Organizer. The system file will be updated with *.scu file information.

Preference Parameter	Default	Affects
AutoSaveBefore	off	If modified diagrams should be saved on file without user confirmation when selecting any of the commands *Analyze*, *Make*, *SDL Overview* and *Convert to PR/MP*.
BuildScript	".bld"	File name extension used for build script files.
Capitalisation	AsDefined	The default value for the code generation option *Capitalization*.
ChangeBars	off	If change bars should be created when SDL diagrams are edited.
ClearLog	on	Clear text in Organizer Log before running the Analyzer.
CoderBufferInSDL	"None"	The default value for the encoding used for ASN.1 Buffer in SDL (see *ASN.1 encode/decode parameter*).
CompileAndLink	on	The default value for the code generation option *Compile & link*.
DefaultCompiler	**On UNIX:** gcc **On Windows:** Microsoft	The default compiler to be used.
DefaultSimulator	"SCTADEBCOM"	The default value for the compile and link option *Standard kernel* when generating a simulator.
DefaultExplorer	"SCTAVALIDATOR"	The default value for the link option *Standard kernel* when generating an SDL Explorer.

Preference Parameter	Default	Affects
EchoAnalyzerCommands	off	The default value for the Analyzer option *Echo Analyzer commands*.
ErrorLimit	30	The default value of the Analyze parameter *Error limit*.
ExpandPR	off	The value of the Convert to PR option *Expand included PR files*.
ExpressionLimit	0	The value of the Analyze parameter *Log expressions deeper than*.
FileNamePrefix	“”	The default value for the code generation option *File name prefix*.
FilterCommand	“”	*Filter command* to invoke before Analyzer stages.
GenerateASN1Coder	off	The default value for the code generation option *Generate ASN.1 coder*.
GenerateCode	on	The default value for the code generation option *Analyze & generate code*.
GenerateEnvFunctions	off	The default value for the code generation option *Generate environment functions*.
GenerateEnvHeader	off	The default value for the code generation option *Generate environment header file*.
GeneratePrefixedNames	off	The default value for the code generator option *Generate prefixed names in ifc file*. Note: This preference is not recognized by Targeting Expert.

Preference Parameter	Default	Affects
GenerateSDLCoder	off	The default value for the code generation option *Generate SDL coder*.
GenerateSignalNumbers	off	The default value for the code generation option *Generate signal number file*.
GenericCommand	**On UNIX:** ".txt "ls %f"" **In Windows:** ".txt "notepad %f""	A list of file extensions and the corresponding command to be performed when editing a generic document. See "Generic Document" on page 40 in chapter 2, *The Organizer*. If used on Windows with full path some extra escape characters must be inserted. As an example to use Microsoft Word as an application for documents with the .doc extension the preference must be set to " .doc "\"C:\\Program Files\\Microsoft Office\\Office10\\WinWord.exe\" %f" " Note that when specifying paths the path delimiter must be entered twice, also if the path contains spaces the path must be enclosed by the \" characters. You can use a list of different file extensions and applications. An example is " .txt "notepad %f" .fm "c:\\frame\\frame.exe %f" .doc "\"c:*path with space*\\ww.exe\" %f" ".
HeaderFileSuffix	".h"	File name extension used to recognize C/C++ header files.

Preference Parameter	Default	Affects
IncludeSdtRef	on	Include SDT References when generating code from SDL.
IDLSuffix	**UNIX only:** .idl	File suffix used for IDL files.
IgnoreHidden	on	Ignore hidden symbols during analyze.
IncludeOptionalFields	off	The default value for the Analyze option *Include optional fields in make operator*.
Kernel	“SCTADEBCOM”	The default value for the compile and link option *Standard kernel*.
LicenseInfo	on	If this preference is on, when failing to get a license during start-up, then information about license holders and number of available licenses is presented in the command line window.
LogFilter		What Analyzer messages are shown in the log window.
LogNavigationLevel	Error	The navigation between messages in the log window.
MacroExpansion	off	The default value for the Analyze option *Macro expansion*.
MakefileMode	Generate	The default value for which option to select from the compile and link options • *Generate makefile* • *Generate makefile and use template* • *Use makefile*

Preference Parameter	Default	Affects
MissingAnswerValues-Control	on	The default value for the Analyze option *Check missing answer values*
MissingElseControl	on	The default value for the Analyze option *Check missing else answers*
ParameterMismatchControl	on	The default value for the Analyze option *Check parameter mismatch*
ExternalTypeFreeControl	on	The default value for the Analyze option *External types should call GenericFree*
MoveWindow	off	The size and position of the Organizer window when a system file is opened. That is, if the size and position will be the same as when that system file was last saved.
OptionalParamControl	on	The default value for the Analyze option *Check optional parameters*.
OutputControl	on	The default value for the Analyze option *Check output semantics*.
PathLineLength	0	How many characters that a file or directory may be, before being split into several lines in the Organizer window. 0 = no limit.
PrefixType	Full	The default value for the code generation option *Prefix*.
QBAddExisting	Hide	The presence of the *Add Existing* quick button.

Preference Parameter	Default	Affects
QBAddNew	Space & Show	The presence of the *Add New* quick button.
QBAnalyze	Space & Show	The presence of the *Analyze* quick button.
QBGenerateSim	Space & Show	The presence of the *Simulate* quick button.
QBGenerateVal	Show	The presence of the *Explore* quick button.
QBGenerateView	Show	The presence of the *Generate SDL Overview* quick button.
QBHelp	Space & Show	The presence of the *Help* quick button.
QBLog	Space & Show	The presence of the *Organizer Log* quick button.
QBLogAnalyze	Space & Show	The presence of the Organizer Log *Analyze* quick button.
QBLogClear	Show	The presence of the Organizer Log *Clear Log* quick button.
QBLogClose	Space & Show	The presence of the Organizer Log *Close* quick button.
QBLogErrorHelp	Show	The presence of the Organizer Log *Help on Error* quick button.
QBLogHelp	Space & Show	The presence of the Organizer Log *Help* quick button.
QBLogMoveDown	Space & Show	The presence of the Organizer Move Down quick button.
QBLogMoveUp	Show	The presence of the Organizer Move Up quick button.
QBLogSave	Show	The presence of the Organizer Log *Save* quick button.

Preference Parameter	Default	Affects
QBLogShowError	Space & Show	The presence of the Organizer Log *Show Error* quick button.
QBLogShowOrganizer	Space & Show	The presence of the Organizer Log *Show Organizer* quick button.
QBMake	Show	The presence of the *Make* quick button.
QBMoveDown	Show	The presence of the *Move Down* quick button.
QBMoveUp	Show	The presence of the *Move Up* quick button.
QBNew	Hide	The presence of the *New* quick button.
QBOpen	Show	The presence of the *Open* quick button.
QBPrint	Space & Show	The presence of the *Print* quick button.
QBSave	Show	The presence of the *Save* quick button.
QBSearch	Space & Show	The presence of the *Search* quick button.
QBTogglePages	Show	The presence of the Show Pages quick button.
QBXref	Show	The presence of the *Generate Cross References* quick button.
QBZoomIn	Show	The presence of the *Increase Scale* quick button.
QBZoomOut	Space & Show	The presence of the *Decrease Scale* quick button.
ReferenceControl	on	The default value for Analyze option *Check references*.

Preference Parameter	Default	Affects
SDLPRFileSuffix	.pr	File suffix used for SDL PR files.
SemanticControl	on	The default value for Analyze option *Semantic analysis*.
Separation	No	The default value for the code generation option *Separation*.
ShowAddExisting	on	If an existing document should be opened in an editor when added.
ShowDashed	on	The default value for the view option *Dashed diagrams*.
ShowDependencies	on	The default value for the view option *Dependency symbols*.
ShowDirectories	off	The default value for the view option *File directories*.
ShowFileName	on	The default value for the view option *File names*.
ShowFooter	off	The default value for the view option *Footer file*.
ShowGroups	on	The default value for the view option *CM Groups*.
ShowHeader	off	The default value for the view option *Header file*.
ShowInstances	on	The default value for the view option *Instance diagrams*.
ShowLinkFile	off	The default value for the view option *Link file*.
ShowLinks	on	The default value for the view option *Association symbols*.

Preference Parameter	Default	Affects
ShowLogLevel	Warning	Under which circumstances the Organizer Log Window should be raised automatically. This preference is also used by Targeting Expert to avoid the popup message when an Analyzer warning or error is found, see "Configure how to Make the Component" on page 2948 in chapter 59, *The Targeting Expert*.
ShowLongMenus	on	The default value for the view options *Long* / *Short*.
ShowPages	off	The default value for the view option *Page symbols*.
ShowPermissions	on	The default value for the view option *File access permissions*.
ShowSeparators	on	The default value for the view option *Separator symbols*.
ShowSource	on	The default value for the view option *Source directory*.
ShowSystemFile	on	The default value for the view option *System file*.
ShowTarget	off	The default value for the view option *Target directory*.
ShowTypeName	off	The default value for the view option *Type names*.
ShowVirtuality	on	The default value for the view option *Virtuality*.
SourceDirectory	""	The default *Source directory* when creating a new system.

Preference Parameter	Default	Affects
StandardKernel	on	The default value for the compile and link option *Standard kernel*.
Statusbar	on	The default value for the view option *Status bar*.
SuppressLevel	5	The number of issued identical Organizer messages before they are suppressed in the Organizer Log window.
SyntaxControl	on	The default value for Analyze option *Syntactic analysis*.
TargetDirectory	""	The default *Target directory* when creating a new system.
TargetLanguage	Cbasic	The default value for the code generation option *Code generator*.
TerminateAnalyzer	off	The default value for the analysis option *Terminate Analyzer when done*.
Toolbar	on	The default value for the view option *Tool bar*.
TrailingParamControl	on	The default value for the Analyze option *Check trailing parameters*.
TreeRepresentation	List	The default value for the view options *Indented list* / *Vertical tree*.
UpperCase	off	The default value of the Convert to PR option *Write reserved words in* (off means lower case).

Preference Parameter	Default	Affects
UsageControl	on	The default value for Analyze option *Check unused definitions*.
UserKernel	""	The default value for the compile and link option *Use kernel in directory*.
UserMakefile	""	The default value for the compile and link option *Use makefile*.
UserTemplate	""	The default value for the compile and link option *Generate makefile and use template*.
WelcomeWindow	on	If a Welcome window should be displayed at start-up of the Organizer.
WordSuffix	".doc"	File name extension used to recognize Microsoft Word files. **Windows only**, see **Note: on page 241**.
XCodeGenerator	"X"	The name of the code generator that appears last in the option menu *Code generator* in the Analyze and generate code options. **UNIX only**, see **Note: on page 241**.
XRef	on	The default value for Analyze option *Generate a cross reference file*.

Help Viewer Preferences

Prefix: Help*

Preference Parameter	Default	Affects
HelpDirectory	""	What help root directory to use for on-line help files. Useful if the on-line help files is available on a network server. See "Configuring the Help Environment" on page 298 in chapter 4, *Managing Preferences* for more information.
HelpViewer	Firefox	The choice of help viewer. "Default Web Browser" (Firefox, Netscape or Internet Explorer) is the default choice **on Windows**.
FirefoxCommand	"firefox"	What command the tools use when starting the help viewer when the preference parameter *HelpViewer* is specified as Firefox.
NetscapeCommand	"netscape"	What command the tools use when starting the help viewer when the preference parameter *HelpViewer* is specified as Netscape.
InternetExplorerCommand	"iexplorer"	What command the tools use when starting the help viewer when the preference parameter *HelpViewer* is specified as Internet Explorer. This is useful on the UNIX platform when the Internet Explorer is not available via your PATH variable.

Preference Manager Preferences

Prefix: Preference*

Preference Parameter	Default	Affects
Statusbar	on	The presence of a status bar in the window.

Preference Parameter	Default	Affects
Toolbar	on	The presence of a tool bar in the window.

Print Preferences

Prefix: Print*

Preference Parameter	Default	Affects
BackwardReferences	on	Print backward paper page references in diagrams.
BlackAndWhite	off	The color used when printing symbols and lines.
DateType	ISO	The date format in printouts.
DestinationFormat	PSFile	The default *Format* option in the Print dialog.
FooterFile	off	The default value for the option *Footer file* in the Print dialog.
ForwardReferences	on	Print forward paper page references in diagrams.
Frame*PaperFormat	USLetter	The default value for the paper format used when printing to FrameMaker, Interleaf or EPS.
Frame*UserDefined-Width	210	The default value for the paper width when the Frame*PaperFormat preference is set to UserDefined.
Frame*UserDefined-Height	297	The default value for the paper height when the Frame*PaperFormat preference is set to UserDefined.
Frame*MarginUpper	25	The default value for the upper margin when printing to FrameMaker, Interleaf or EPS.
Frame*MarginLower	35	The default value for the lower margin when printing to FrameMaker, Interleaf or EPS
Frame*MarginLeft	25	The default value for the left margin when printing to FrameMaker, Interleaf or EPS

Preference Parameter	Default	Affects
Frame*MarginRight	27	The default value for the right margin when printing to FrameMaker, Interleaf or EPS
FrameMakerCommand	“maker”	The default *Execute* command in the Print dialog when *Format* is specified as *Import into FrameMaker*. **UNIX only**, see **Note: on page 241**.
HeaderFile	““	The default value for the option *Header file* in the Print dialog.
HeaderTextHeight	12 (pt)	The size of the font face used in headers and footers.
Landscape	off	The default value for the option *Orientation* in the Print dialog (off means landscape).
MarginLeft	10 (mm)	The default value for the parameter *Left margin* in the Print Margins dialog.
MarginLower	25 (mm)	The default value for the parameter *Lower margin* in the Print Margins dialog.
MaxPageReferences	20	Max number of page references at one place.
MarginRight	10 (mm)	The default value for the parameter *Right margin* in the Print Margins dialog.
MarginUpper	42 (mm)	The default value for the parameter *Upper margin* in the Print Margins dialog.
MaxTextLineLength	80	How long a line in a text file can be before it is divided/wrapped, resulting in several lines in the printout. A value of zero means no wrapping. Lines are usually divided at word boundaries.

Preference Parameter	Default	Affects
MinimumLineWidth	3 (1/10 pt)	The minimum thickness applied on lines in the printout.
OnlyChaptersInTOC	off	If only chapter symbols, or chapter symbols and diagrams and documents will be visible in the table of contents.
OrganizerView	on	The default value for the option *Organizer View* in the Organizer *Print* dialog.
PageMarkers	off	The default value for the option *Page markers* in the *Print* dialog.
PaperFormat	A4	The default value for the option *Paper format* in the *Print* dialog. The preference variable Print*PaperFormat will be set to USLetter by default during installation if the current local time is within GMT - 05:00 and GMT - 09:00. (English version only) This is done by setting the preference variable Print*PaperFormat within the file sdt.ini (Windows) or .sdtpref (UNIX) located in .../locale/english/sdt. To change this to default value A4, remove the file or use the Preference manager.
Poster	off	Print header (footer) for each logical page instead of for each physical page.
PrintChapter	on	Print text associated with chapter symbols.
PrintCollapsed	on	If collapsed texts from text, comment and text extension symbols are printed in full on a separate page or not.

Preference Parameter	Default	Affects
PrinterCommand	"lpr -h -r" (**In Windows**, the lpr command is available in the distribution.)	The default *Execute* command in the Print dialog when *Format* is specified as *One PostScript File*.
RemoveTempFiles	on	If temporary files should be removed after printing is done.
Scale	100 (%)	The default value for the parameter *Scale* in the Print dialog.
TableOfContents	off	The default value for the option *Table of Contents* in the Organizer *Print* dialog.
TextHeight	10 (mm)	The font size used when printing expanded text symbols.
ToCShowPages	off	The presence of page numbers in generated table of contents.
UserDefinedHeight	297 (mm)	The default value for the parameter *User paper height* in the Print Margins dialog.
UserDefinedWidth	210 (mm)	The default value for the parameter *User paper width* in the Print Margins dialog.
Word*ImageFormat	Normal.dot	The value for the *Image format (Windows Only)* used when printing to Word. When using Normal.dot, the image size will be calculated from the available print area in a new created Word document.
Word*UserDefined-Height	247 (mm)	The default value for the image height parameter in the Size dialog using the *User Defined size* option.
Word*UserDefined-Width	160 (mm)	The default value for the image height parameter in the Size dialog using the *User Defined size* option.

Text Editor Preferences

Prefix: TE*

Preference Parameter	Default	Affects
AlwaysNewWindow	off	If a new window should be opened or not when opening a text document.
PrintFontFamily	“Courier”	The font face used when printing text.
ShowLinks	on	If endpoints and links should be shown.
StateMatrixShowSelected	on	Decides if processes specified with StateMatrixFilter should be hidden or shown.
StateMatrixSortSignals	on	If signals in state matrices should be sorted.
StateMatrixSortStates	on	If states in state matrices should be sorted.
StateMatrixCalls	off	If a state matrix with procedure call transition information should be shown or not.
StateMatrixFilter	““	Which processes that state matrices will be generated for.
StateMatrixNextstates	on	If a state matrix with nextstate transition information should be shown or not.
StateMatrixOutputs	off	If a state matrix with output transition information should be shown or not.
StateMatrixPageNumbers	on	If a state matrix with page numbers as transition information should be shown or not.
Statusbar	on	The presence of a status bar in the window.
Toolbar	on	The presence of a tool bar in the window.
TextTemplateDirectory	““	Directory for common text file templates.

The SDL Suite Specific Preferences

These preferences are only available for the SDL Suite tools.

SDL Editor Preferences

Prefix: Editor*

The Beige color used has the RGB values: (255, 249, 242)

Preference Parameter	Default	Affects
AdditionalHeadingOnlyOn-FirstPage	off	If the Additional Heading symbol should be shown only on the first page.
AlwaysNewWindow	off	If a new window should be opened or not when opening an SDL diagram. See "Always new Window" on page 2018 in chapter 43, *Using the SDL Editor*.
BlockSubstructureSymbol*Use	on	If the symbol should be available in the symbol menu.
BlockSubstructureSymbol*Color	Black	The color of the symbol.
BlockSubstructureSymbol*Fill-Color	Beige	The fill color of the symbol.
BlockSymbol*Use	on	If the symbol should be available in the symbol menu.
BlockSymbol*Color	Black	The color of the symbol.
BlockSymbol*FillColor	Beige	The fill color of the symbol.
BlockTypeSymbol*Use	on	If the symbol should be available in the symbol menu.
BlockTypeSymbol*Color	Black	The color of the symbol.
BlockTypeSymbol*FillColor		The fill color of the symbol.
ChannelSubstructureSymbol*Use	on	If the symbol should be available in the symbol menu.

Preference Parameter	Default	Affects
ChannelSubstructureSymbol*Color	Black	The color of the symbol.
ChannelSubstructureSymbol*FillColor	Beige	The fill color of the symbol.
ClassSymbol*Use	on	If the symbol should be available in the symbol menu.
ClassSymbol*Color	Black	The color of the symbol.
ClassSymbol*FillColor	Beige	The fill color of the symbol.
CommentLeftSymbol*Use	off	If the symbol should be available in the symbol menu.
CommentLeftSymbol*Color	Black	The color of the symbol.
CommentLeftSymbol*FillColor	Beige	The fill color of the symbol.
CommentSymbol*Use	on	If the symbol should be available in the symbol menu.
CommentSymbol*Color	Black	The color of the symbol.
CommentSymbol*FillColor	Beige	The fill color of the symbol.
CompletionMaxPages	10	The maximum number of initial diagram pages searched by the word completion operation.
ConnectorSymbol*Use	on	If the symbol should be available in the symbol menu.
ConnectorSymbol*Color	Black	The color of the symbol.
ConnectorSymbol*FillColor	Beige	The fill color of the symbol.
ContinuousSignalSymbol*Use	on	If the symbol should be available in the symbol menu.
ContinuousSignalSymbol*Color	Black	The color of the symbol.
ContinuousSignalSymbol*FillColor	White	The fill color of the symbol.
CreateRequestSymbol*Use	on	If the symbol should be available in the symbol menu.

Preference Parameter	Default	Affects
CreateRequestSymbol*Color	Black	The color of the symbol.
CreateRequestSymbol*FillColor	Beige	The fill color of the symbol.
DecisionSymbol*Use	on	If the symbol should be available in the symbol menu.
DecisionSymbol*Color	Black	The color of the symbol.
DecisionSymbol*FillColor	Beige	The fill color of the symbol.
FontText*ScreenFontFamily	“Helvetica”	The font face to use when displaying text on screen.
FontText*PrintFontFamily	“Helvetica”	The font face used when printing text.
FontText*TextSymbolFontFamily	“Helvetica“	The font face to use when displaying text in text symbols on screen **and** in print.
FontText*TextHeight	9 (pt)	The height of text in symbols (except kernel heading symbols and page name symbols) on flow diagrams, and text symbols and signal list symbols on interaction diagrams.
FontText*NameTextHeight	12 (pt)	The height of text in kernel heading symbols, page name symbols and text objects on interaction diagrams (exception from text symbols and signal list symbols). See “Default Font Sizes” on page 1969 in chapter 43, *Using the SDL Editor*.
FontText*TaskSymbolLeftAligned	off	If the text in the task, procedure call, macro call, create request and save symbol should be left aligned.
FontText*MinimumTextUpdateDelay	0 (milliseconds)	The minimum time delay to update text in symbols.

Preference Parameter	Default	Affects
FontText*BlinkingTextCursor	on	If the text cursor (vertical bar) should blink in the drawing area.
GateSymbol*Use	on	If the symbol should be available in the symbol menu.
GateSymbol*Color	Black	The color of the symbol.
GlobalSymbolColor*Color	Black	The color of all symbols
GlobalSymbolColor*UseGlobal-Color	off	If the color for all symbols should be used.
GlobalSymbolColor*FillColor	White	The fill color of all symbols.
GlobalSymbolColor*UseGlobal-FillColor	off	If the fill color for all symbols should be used.
GrammarHelp	off	If the Grammar Help window is to be opened automatically at start-up.
InputLeftSymbol*Use	off	If the symbol should be displayed left-oriented or not.
InputLeftSymbol*Color	Black	The color of the symbol.
InputLeftSymbol*FillColor	Beige	The fill color of the symbol.
InputSymbol*Use	on	If the symbol should be available in the symbol menu.
InputSymbol*Color	Black	The color of the symbol.
InputSymbol*FillColor	Beige	The fill color of the symbol.
InteractionPage*GridHeight	5 (mm)	The vertical grid on interaction pages.
InteractionPage*GridWidth	5 (mm)	The horizontal grid on interaction pages.
InteractionPage*MarginLeft	15 (mm)	The distance between the The Frame and the left boundary of the drawing area.

Preference Parameter	Default	Affects
InteractionPage*MarginLower	15 (mm)	The distance between the The Frame and the lower boundary of the drawing area.
InteractionPage*MarginRight	15 (mm)	The distance between the The Frame and the right boundary of the drawing area.
InteractionPage*MarginUpper	15 (mm)	The distance between the The Frame and the upper boundary of the drawing area.
InteractionPage*SymbolHeight	20 (mm)	The default height of symbols on interaction diagrams.
InteractionPage*SymbolWidth	30 (mm)	The default width of symbols on interaction diagrams.
MacroCallSymbol*Use	on	If the symbol should be available in the symbol menu.
MacroCallSymbol*Color	Black	The color of the symbol.
MacroCallSymbol*FillColor	Beige	The fill color of the symbol.
MacroInletSymbol*Use	on	If the symbol should be available in the symbol menu.
MacroInletSymbol*Color	Black	The color of the symbol.
MacroInletSymbol*FillColor	Beige	The fill color of the symbol.
MacroOutletSymbol*Use	on	If the symbol should be available in the symbol menu.
MacroOutletSymbol*Color	Black	The color of the symbol.
MacroOutletSymbol*FillColor	Beige	The fill color of the symbol.
OperatorSymbol*Use	on	If the Operator reference symbol should be available or not in the symbol menu. (The symbol is an SDL Suite specific extension to Z.100.)
OperatorSymbol*Color	Black	The color of the symbol.

Preference Parameter	Default	Affects
OperatorSymbol*FillColor	Beige	The fill color of the symbol.
OutputLeftSymbol*Use	off	If the symbol should be displayed left-oriented or not.
OutputLeftSymbol*Color	Black	The color of the symbol.
OutputLeftSymbol*FillColor	Beige	The fill color of the symbol.
OutputSymbol*Use	on	If the symbol should be available in the symbol menu.
OutputSymbol*Color	Black	The color of the symbol.
OutputSymbol*FillColor	Beige	The fill color of the symbol.
Page*AutoNumber	on	If added SDL pages should be autonumbered (1, 2,... N). See "The Autonumbered Option" on page 2036 in chapter 43, *Using the SDL Editor*.
Page*Height	230 (mm)	The height of a new page.
Page*Width	190 (mm)	The width of a new page.
Page*EnumerationCharacter	_	When two pages are given the same name X, their default names will be X_1 and X_2. The character used for enumeration can be any of '_', '.', '#', '@', '$' or '|'.
PrintZ100Symbols	off	Printing SDL symbols with an appearance that complies to Z.100.
PriorityInputLeftSymbol*Use	off	If the symbol should be displayed left-oriented or not.
PriorityInputLeftSymbol*Color	Black	The color of the symbol.
PriorityInputLeftSymbol*FillColor	Beige	The fill color of the symbol.

Preference Parameter	Default	Affects
PriorityInputSymbol*Use	on	If the symbol should be available in the symbol menu.
PriorityInputSymbol*Color	Black	The color of the symbol.
PriorityInputSymbol*FillColor	Beige	The fill color of the symbol.
ProcedureCallSymbol*Use	on	If the symbol should be available in the symbol menu.
ProcedureCallSymbol*Color	Black	The color of the symbol.
ProcedureCallSymbol*FillColor	Beige	The fill color of the symbol.
ProcedureReturnSymbol*Use	on	If the symbol should be available in the symbol menu.
ProcedureReturnSymbol*Color	Black	The color of the symbol.
ProcedureReturnSymbol*FillCol-or	Beige	The fill color of the symbol.
ProcedureStartSymbol*Use	on	If the symbol should be available in the symbol menu.
ProcedureStartSymbol*Color	Black	The color of the symbol.
ProcedureStartSymbol*FillColor	Beige	The fill color of the symbol.
ProcedureSymbol*Use	on	If the symbol should be available in the symbol menu.
ProcedureSymbol*Color	Black	The color of the symbol.
ProcedureSymbol*FillColor	Beige	The fill color of the symbol.
ProcessPage*GridHeight	5 (mm)	The vertical grid on flow dia-grams.
ProcessPage*GridWidth	5 (mm)	The horizontal grid on flow dia-grams.
ProcessPage*MarginLeft	0 (mm)	The distance between the The Frame and the left boundary of the drawing area.

Preference Parameter	Default	Affects
ProcessPage*MarginLower	0 (mm)	The distance between the The Frame and the lower boundary of the drawing area.
ProcessPage*MarginRight	0 (mm)	The distance between the The Frame and the right boundary of the drawing area.
ProcessPage*MarginUpper	0 (mm)	The distance between the The Frame and the upper boundary of the drawing area.
ProcessPage*SymbolHeight	10 (mm)	The default height for symbols on flow pages. (The width / height relationship is fixed to 2:1.)
ProcessSymbol*Use	on	If the symbol should be available in the symbol menu.
ProcessSymbol*Color	Black	The color of the symbol.
ProcessSymbol*FillColor	Beige	The fill color of the symbol.
ProcessTypeSymbol*Use	on	If the symbol should be available in the symbol menu.
ProcessTypeSymbol*Color	Black	The color of the symbol.
ProcessTypeSymbol*FillColor	Beige	The fill color of the symbol.
SaveSymbol*Use	on	If the symbol should be available in the symbol menu.
SaveSymbol*Color	Black	The color of the symbol.
SaveSymbol*FillColor	Beige	The fill color of the symbol.
SDLTraceColor	yellow	The temporary background color to use for visited SDL symbols during SDL trace.
Scale	100 (%)	The scale when displaying a diagram on screen.

Preference Parameter	Default	Affects
ScreenZ100Symbols	off	Displaying SDL symbols on screen with an appearance that complies to Z.100.
ServiceSymbol*Use	on	If the symbol should be available in the symbol menu.
ServiceSymbol*Color	Black	The color of the symbol.
ServiceSymbol*FillColor	Beige	The fill color of the symbol.
ServiceTypeSymbol*Use	on	If the symbol should be available in the symbol menu.
ServiceTypeSymbol*Color	Black	The color of the symbol.
ServiceTypeSymbol*FillColor	Beige	The fill color of the symbol.
ShowGrid	off	If grid points should be shown on screen.
ShowLinks	on	If endpoints and links should be shown.
ShowPageBreaks	on	If printer page breaks should be visible or not on the screen.
ShowSignalDeclaration	on	If the signal declaration should be shown in the message area for a selected signal name.
SignalDictionary*Use	off	If a Signal Dictionary window should be opened automatically upon start of the SDL Editor.
SignalDictionary*All	off	If the Signal Dictionary option All should be enabled by default.
SignalDictionary*Down	on	If the Signal Dictionary option Down should be enabled by default.
SignalDictionary*External	off	If the Signal Dictionary option External should be enabled by default.

Preference Parameter	Default	Affects
SignalDictionary*MSC	off	If the Signal Dictionary option MSC should be enabled by default.
SignalDictionary*This	off	If the Signal Dictionary option This should be enabled by default.
SignalDictionary*Up	on	If the Signal Dictionary option Up should be enabled by default.
StartSymbol*Use	on	If the symbol should be available in the symbol menu.
StartSymbol*Color	Black	The color of the symbol.
StartSymbol*FillColor	Beige	The fill color of the symbol.
StateSymbol*Use	on	If the symbol should be available in the symbol menu.
StateSymbol*Color	Black	The color of the symbol.
StateSymbol*FillColor	Beige	The fill color of the symbol.
Statusbar	on	The presence of a status bar in the SDL Editor window.
StopSymbol*Use	on	If the symbol should be available in the symbol menu.
StopSymbol*Color	Black	The color of the symbol.
StopSymbol*FillColor	White	The fill color of the symbol.
SymbolMenu	on	The presence of a symbol menu in the SDL Editor window.
SyntaxCheck	on	Enables or disables the syntactic checking performed by the SDL Editor at editing time.
SystemTypeSymbol*Use	on	If the symbol should be available in the symbol menu.
SystemTypeSymbol*Color	Black	The color of the symbol.

Preference Parameter	Default	Affects
SystemTypeSymbol*FillColor	Beige	The fill color of the symbol.
TabEqualsCompletion	on	If the tab key invokes word completion or produces a tab character.
TaskSymbol*Use	on	If the symbol should be available in the symbol menu.
TaskSymbol*Color	Black	The color of the symbol.
TaskSymbol*FillColor	Beige	The fill color of the symbol.
TemplateFile	“sdt.tpl”	The name of the file containing the grammar templates used by the Grammar Help window.
TextExtensionLeftSymbol*Use	off	If the flipped text extension symbol should be available in the symbol menu.
TextExtensionLeftSymbol*Color	Black	The color of the flipped text extension symbol.
TextExtensionLeftSymbol*FillColor	Beige	The fill color of the flipped text extension symbol.
TextExtensionSymbol*Use	on	If the symbol should be available in the symbol menu.
TextExtensionSymbol*Color	Black	The color of the symbol.
TextExtensionSymbol*FillColor	Beige	The fill color of the symbol.
TextReferenceSymbol*Use	on	If the symbol should be available in the symbol menu.
TextReferenceSymbol*Color	Black	The color of the symbol.
TextReferenceSymbol*FillColor	Beige	The fill color of the symbol.
TextSymbol*Use	on	If the symbol should be available in the symbol menu.
TextSymbol*Color	Black	The color of the symbol.
TextSymbol*FillColor	Beige	The fill color of the symbol.

Preference Parameter	Default	Affects
TextWindow	on	The presence of a text window in the SDL Editor window.
TextualSyntaxCheck	on	Textual syntax check in symbols.
TidyUp*MaxCommentLine-Length	22	Maximum number of characters per line in comment symbols after tidy up
TidyUp*TransNewPage	on	Start each transition on a new page
TidyUp*ExtraSpace	on	Insert extra space after a symbol attached to a large comment or text extension, to avoid overlapping any comment or text extension attached to the next symbol.
TidyUp*FormatInputOutput	on	Format text in input and output symbols using newline characters and text extension symbols, to try to avoid text outside the symbol.
TidyUp*MakeCommentSymbols	on	Replace textual comments with comment symbols.
TidyUp*CommentsLeft	on	Place comment symbols to the left of the symbols that they are attached to.
TidyUp*SmartAutoLabels	on	Generate automatic names for all labels. The names are generated from the closest available state-name. If the preference is off the names for the already existing labels will be preserved and new generated labels will have names LblNNN.
TidyUp*NoStateExtension	on	Avoid attaching text extension symbols to state symbols.

Preference Parameter	Default	Affects
TidyUp*NoDecisionExtension	on	Avoid attaching text extension symbols to decision symbols.
TidyUp*SortStates	off	Sort states and transitions in alphabetical order.
Toolbar	on	The presence of a tool bar in the SDL Editor window.
TransitionOptionSymbol*Use	on	If the symbol should be available in the symbol menu.
TransitionOptionSymbol*Color	Black	The color of the symbol.
TransitionOptionSymbol*FillColor	Beige	The fill color of the symbol.

OM/SC/HMSC/MSC/DP Editor Preferences

Prefix: OME*

Preference Parameter	Default	Affects
AlwaysEndpointClass	on	If an endpoint should be automatically added on new class symbols.
AlwaysEndpointObject	off	If an endpoint should be automatically added on a new object symbol.
AlwaysNewWindow	off	If a new window should be opened or not when opening a diagram. See "Always new Window" on page 1677 in chapter 39, *Using Diagram Editors*.
AutoNumberPages	on	If added pages should be autonumbered (1, 2,... N). See "Autonumbered" on page 1681 in chapter 39, *Using Diagram Editors*.
BlinkingTextCursor	on	If the text cursor (vertical bar) should blink in the drawing area.
DPComponentSymbolColor	Black	The color of the symbol.
DPNodeSymbolColor	Black	The color of the symbol.
DPObjectSymbolColor	Black	The color of the symbol.
DPThreadSymbolColor	Black	The color of the symbol.
HMSCConditionSymbolColor	Black	The color of the symbol.
HMSCConnectionSymbolColor	Black	The color of the symbol.
HMSCReferenceSymbolColor	Black	The color of the symbol.
HMSCStartSymbolColor	Black	The color of the symbol.
HMSCStopSymbolColor	Black	The color of the symbol.
MinimumTextUpdateDelay	0 (milliseconds)	The minimum time delay to update text in symbols.

Preference Parameter	Default	Affects
MSCHorizontalSpacing	25 (mm)	The horizontal spacing unit between instances. See "Spacing" on page 1677 in chapter 39, *Using Diagram Editors*.
MSCInstanceRuler	on	The presence of an Instance Ruler in the MSC Editor window.
MSCPrintInstanceRuler	on	If an Instance Ruler should be included on the printed pages.
MSCVerticalSpacing	3 (mm)	The vertical spacing unit between instances. See "Spacing" on page 1677 in chapter 39, *Using Diagram Editors*.
NameTextHeight	12 (pt)	The height of text in names of symbols.
OMClassSymbolColor	Black	The color of the symbol.
OMObjectSymbolColor	Black	The color of the symbol.
PageHeight	230 (mm)	The height of a new page.
PageWidth	190 (mm)	The width of a new page.
PrintFontFamily	"Helvetica"	The font face used when printing text.
Scale	100 (%)	The scale when displaying a diagram on screen.
ScreenFontFamily	"Helvetica"	The font face to use when displaying text on screen.
SCStartSymbolColor	Black	The color of the symbol.
SCStateSymbolColor	Black	The color of the symbol.
SCStopSymbolColor	Black	The color of the symbol.
ShowGrid	off	If grid points should be shown on screen.
ShowInstanceComposition	on	If the Instance decomposition should be displayed.

Preference Parameter	Default	Affects
ShowInstanceKind	on	If the Instance kind should be displayed.
ShowInstanceName	on	If the Instance name should be displayed.
ShowLinks	on	If endpoints and links should be shown.
ShowMessageName	on	If the Message name should be displayed.
ShowMessageParameters	on	If the Message parameters should be displayed.
ShowPageBreaks	on	If printer page breaks should be visible or not on the screen.
Statusbar	on	The presence of a status bar in the editor window.
SymbolMenu	on	The presence of a symbol menu in the editor window.
TextHeight	9 (pt)	The height of text in symbols (except heading symbols and page name symbols) and line attributes, except for names of symbols.
TextSymbolColor	Black	The color of the symbol.
TextWindow	on	The presence of a text window in the editor window.
Toolbar	on	The presence of a tool bar in the editor window.

Index Viewer Preferences

Prefix: Index*

Preference Parameter	Default	Affects
FilterTypes	""	Which SDL entities to be hidden by default in the *Filter Types* dialog. The entities should be entered as a number of text strings, separated by spaces.
FilterDiagrams	""	Which SDL entities that will be visible.
FilterUses	""	Which kind of entity use that will be visible.
IndexAppearance	Detailed	The layout of SDL entity information. The alternatives are Compact, VerticalCompact, HorizontalDetailed and Detailed.
NameFirst	on	If entities will be sorted by name or by type and name.
ShowSelectedDiagrams	on	If diagrams specified with *FilterDiagrams* should be shown or hidden.
ShowSelectedTypes	on	If types specified with *FilterTypes* should be shown or hidden.
ShowSelectedUses	on	If uses specified with *FilterUses* should be shown or hidden.
Statusbar	on	The presence of a status bar in the window.
Toolbar	on	The presence of a tool bar in the window.

Type Viewer Preferences

Prefix: Type*

Preference Parameter	Default	Affects
ShowInstanceSymbols	on	The default value for the *Instance symbols* option defined in the *List Options*.
ShowNameText	on	The default value for the *Name* option defined in the *Symbol Options*.

Preference Parameter	Default	Affects
ShowQualifierText	off	The default value for the *Qualifier* option defined in the *Symbol Options*.
ShowTypeText	on	The default value for the *Type* option defined in the *Symbol Options*.
Statusbar	on	The presence of a status bar in the window.
Toolbar	on	The presence of a tool bar in the window.

Coverage Viewer Preferences

Prefix: Cover*

Preference Parameter	Default	Affects
AboveThreshold	on	Which option to select by default between *Show symbols executed >= threshold* and *Show symbols executed <= threshold* (on means >=).
FillSymbols	on	Whether to fill symbols or not when displaying a coverage tree.
LogThickness	off	Which option to select by default between *Normal line thickness* and *Logarithmic line thickness* (off means normal).
MaxLineThickness	60 (1/10 mm)	The default value for the parameter *Max line thickness*.
ShowTransitions	on	Which option to select by default between *Transition Coverage* and *Symbol Coverage* (on means transition).
ShowVisibilityLine	on	Which option to select by default between *Show visibility line* and *Hide Visibility line* (on means show).
Statusbar	on	The presence of a status bar in the window.
Toolbar	on	The presence of a tool bar in the window.

Preference Parameter	Default	Affects
VisibilityThreshold	0	The default value for the parameter *Threshold*.

SDL Overview Generator Preferences

Prefix: Overview*

Preference Parameter	Default	Affects
DistanceBetweenLines	3 (mm)	The default value for the option *Distance between Lines*.
DistanceBetweenSymbols	10 (mm)	The default value for the option *Distance between Symbols*.
ExpandedDiagramsOnly	on	The default value for the option *Use Expanded Diagrams Only*.
IncludeLines	on	The default value for the option *Include Lines*.
IncludeProcedures	off	The default value for the option *Include Procedures & Operators*.
IncludeStates	off	The default value for the option *Include States*.
Margin	10 (mm)	The default value for the parameter *Margin*.
MinimumSymbolHeight	10	The default value for the option *Minimum Symbol Height*.
MinimumSymbolLength	20	The default value for the option *Minimum Symbol Width*.
NameTextHeight	10 (pt)	The size of textual elements in Overview diagrams. See "Default Font Sizes" on page 1969 in chapter 43, *Using the SDL Editor*.
PrintFontFamily	"Helvetica"	The font face used when printing text.

Preference Parameter	Default	Affects
ScreenFontFamily	“Helvetica”	The font face to use when displaying text on screen.
ShowLineNames	on	The default value for the SDL Editor option *Show line names*.
ShowSignallists	off	The default value for the SDL Editor option *Show signal lists*.
TextHeight	8 (pt)	The size of textual elements in Overview diagrams. See “Default Font Sizes” on page 1969 in chapter 43, *Using the SDL Editor*.

Simulator GUI Preferences

Prefix: SGUI*

Preference Parameter	Default	Affects
ButtonFileName	“def.btns”	What file containing *Button definitions* to load by default.
CommandFileName	“def.cmds”	What file containing *Command definitions* to load by default.
FailOnUnexpectedSignal	off	Makes the line “unexpected” appear in the “command statistics” when running Simulator test scripts. If this preference is set and an unexpected signal is detected before the current signal is checked, then it will be reported.
WatchFileName	“def.vars”	What file containing *Variable definitions* to load by default.
MaxOutputToRetries	50	Maximum number of retries of next-transition and output-to, when output-to fails during simulator script execution.
TextBufferLimit	10000	Maximum number of characters in the simulator UI text log. A value of 0 means “no limit”.

SDL Explorer GUI Preferences

Prefix: VGUI*

Preference Parameter	Default	Affects
ButtonFileName	“val_def.btns”	What file containing *Button definitions* to load by default.
CommandFileName	“val_def.cmds”	What file containing *Command definitions* to load by default.
WatchFileName	“val_def.vars”	What file containing *Variable definitions* to load by default.

Chapter

4

Managing Preferences

This chapter describes the principles for the management of preference parameters in a SDL Suite and TTCN Suite environment. It also describes how to use the Preference Manager to customize the installation.

For a reference to the Preference Manager and the preference parameters, see chapter 3, *The Preference Manager*.

Introduction

Different users are likely to use SDL Suite and TTCN Suite in different ways. Even the same user may use the tools differently in different work areas. The *Preference Manager* allows you to customize the default behavior of the SDL Suite and TTCN Suite tools in order to minimize the number of operations that you have to perform manually after having started a tool.

The Preference Manager allows you to set up the default behavior for the following tools:

- Organizer
- SDL Editor
- Diagram editors
- Text Editor
- Index Viewer
- Type Viewer
- Coverage Viewer
- Overview Generator and Editor
- Simulator Graphical User Interface
- SDL Explorer Graphical User Interface
- Help viewer
- Preference Manager
- Print
- generic SDL Suite and TTCN Suite tool preferences.

Note:

The Preference Manager only handles the SDL Suite-specific tools, and the common tools. The behavior of the TTCN Suite-specific tools cannot be changed by these preferences. For more information, see chapter 29, *Customizing the TTCN Suite (on UNIX)*.

Preferences may be customized for an individual user or may apply to an entire project or even organization that uses the SDL Suite and TTCN Suite tools.

How Preferences Are Managed

Depending on your needs and how you configure your environment, the SDL Suite and TTCN Suite tools handle preference parameters either as:

- *Organization (company) preferences*

  These are parameters that should affect all work that is done with the tools within an organization; for example you might want to prohibit the use of some specific SDL symbols, to set up printout headers and footers with your company's logotype, etc.

- *Project preferences*

  These are parameters that should affect the properties that are specific for a particular project; for instance what *make* and *build* options should be used by the Analyzer and Code Generators within the scope of a project. *Project preferences* can also be regarded as a specialization of the *organization preferences*, specific for a project.

- *User-defined preferences*

  User-defined preferences are parameters that would typically affect the behavior of the tools to have them fulfil the individual user's needs; such as what additional windows to open or not open at start-up of the tool.

- *Default (factory) settings*

  These settings are "burnt-in" into the tools and will be used whenever no other preferences are available.

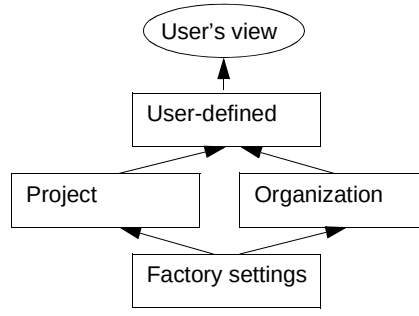


Figure 68: The user's view of the preference parameters

Preferences and Dialog Options

The tools allow you to customize a number of options that define the **default** behavior of a tool, using the Preference Manager (the TTCN Suite specific preferences can be customized as described in chapter 29, *Customizing the TTCN Suite (on UNIX)*). Some of these preferences may be overridden during the current tool session, by assigning a new value to the option. The option's latest value will then be memorized and reused, where possible.

Example 15: Print preferences and options. ––––––––––––––––––––

1. Assume that you have assigned the Print preference parameter *PrinterCommand* the value `lpr` (i.e. send to printer queue).
2. You print from the SDL Editor, change the option *Execute* to `ghostview` (i.e. a PostScript previewer) and click *Print*. Next time you print from the SDL Editor, the Execute option is memorized and reads `ghostview` (until you terminate the SDL Editor session).
3. But, when you print from another tool (the MSC Editor for instance), the *Execute* option is preset to the preferred value, i.e. `lpr`.

––

Preferences and the SDL Suite Diagram Options

The SDL Suite uses your preferred values when you create a new diagram and assign it:

- A width (set with the command *Drawing Size*)
- A height (set with the command *Drawing Size*)
- A viewing scale (set with the command *Set Scale*)

Each individual SDL diagram also stores information about:

- Whether to align symbols and lines to the grid or not
- Whether to check syntax or not when the diagram is edited
- Whether to fix endpoints at frame or not
- Whether to show the Additional Heading symbol only on the first page

  These diagram parameters are set with the SDL Editor command *Diagram Options*.

Each individual MSC also stores information about:

- Whether to show the *instance name* or not
- Whether to show the *instance kind* or not
- Whether to show the *instance composition* or not
- Whether to show the *message name* or not
- Whether to show *message parameters* or not

  These diagram parameters are set with the MSC Editor command *Diagram Options*.

Each individual OM, SC and HMSC diagram also stores information about:

- Whether to align symbols and lines to the grid or not

When you save a diagram, the current diagram parameter values are saved on the diagram file and will be reused next time you open it, **overriding the preference parameters**.

Preferences and System File Options

The tools use your preferred values when creating new system files (the `.sdt` files that are managed by the Organizer).

When you save a system file, the parameters that are related to the system are also stored on the system file, **overriding the preference parameters**:

- The parameters managed by the *Set Directories* dialog
- The parameters managed by the *View Options* dialog
- The parameters managed by the *Set Scale* dialog
- The parameters managed by the *Analyze* dialog
- The parameters managed by the *Make* dialog

See "Options in the System File" on page 193 in chapter 2, *The Organizer* for a reference to the contents of a system file and what parameters are stored on it.

Starting the Preference Manager

> **Note:**
>
> Each individual user should make sure the HOME environment variable is defined in order to take advantage of the Preference Manager. The HOME variable normally denotes the user's home directory.

To start the Preference Manager:

- Select *Preference Manager* from the Organizer's *Tools* menu.

  The Preference Manager will be opened.

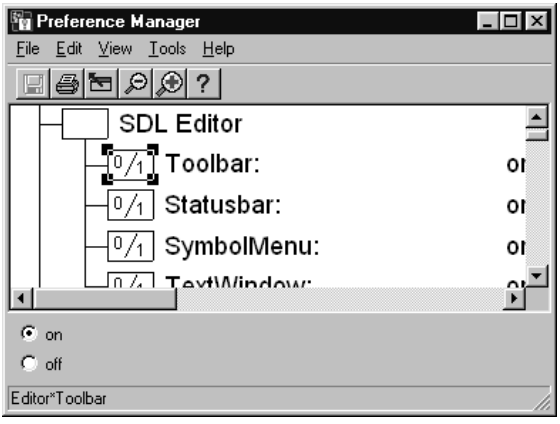


Figure 69: The Preference Manager window

Only one instance of the Preference Manager can be started at a time.

Adjusting a Preference Parameter

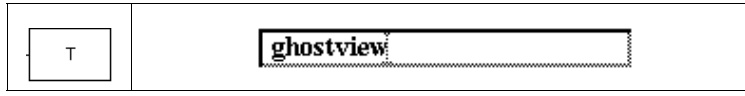
To modify a preference parameter:

1. First, locate the symbol representing the tool it governs (you may have to scroll the tree area up or down to make the symbol visible and to expand it with the *Expand* menu choice from *View Menu*).
2. Once the parameter is located, select it. (The information within parentheses is the current value, the item within brackets shows the source, i.e. user-defined / project / organization).
3. The area at the bottom of the window is updated according to the type of parameter you have selected. The sections below describe how to adjust the preference parameter.

Note:

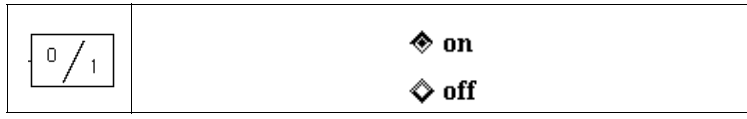
You have the possibility to revert any parameter to its saved value or default value, using the *Set Saved Value* / *Set Default Value* command.

Adjusting a Textual Parameter



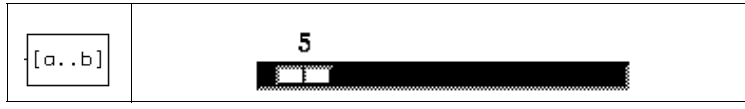
To edit a textual value, simply place the cursor on the text field and enter the text of your choice. No control of what is entered takes place.

Adjusting a Boolean Parameter



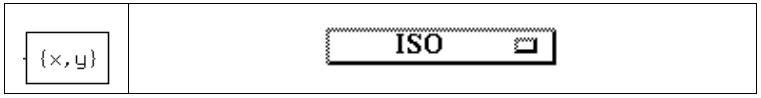
Click *on* to turn an option on.

Adjusting a Ranged Integer



Drag the slider for a coarse adjustment or click the slider bar for a fine adjustment.

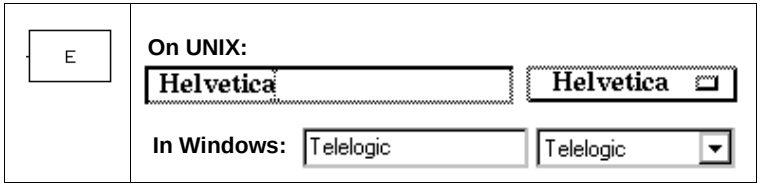
Adjusting an Option Menu



Clicking on the button brings up a pop-up menu where all permissible values are listed. Select the value of your choice.

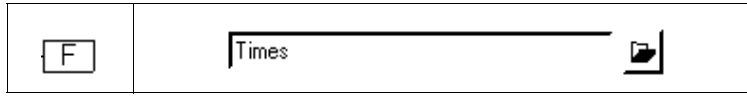
– When you move the mouse pointer over a value in the option menu, a short description of the value is printed in the status bar. Values whose description starts with “(UNIX only)” or “(Windows only)” should only be selected on a UNIX system or a Windows system, respectively.

Adjusting an Option Menu Textual Preference



You may select the preference parameter from a number of predefined values, available on an option menu (see above). As an alternative, you may type in the value into the text field, which grants access to not predefined values.

Adjusting a Font Preference (Windows only)

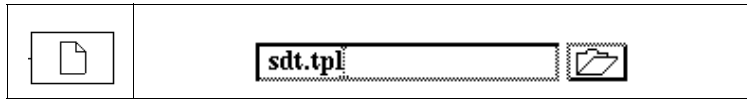


You must select the font by clicking the *folder button*, which opens a Microsoft Windows Font dialog. It is not possible to edit the text field to the left of the folder button.

In the font dialog, all TrueType fonts in the system are listed. It is also possible to select font style and size, but these values have no effect. Only the specified font name is used.

- Any font name can be entered by editing the *Font* text field in the font dialog. This possibility is useful when setting a font for printing that is not available in the system, but it does exist on the printer.
- The SDL Suite will also recognize the three font names “Times”, “Helvetica” and “Courier”. These are translated into the TrueType fonts Times New Roman, Arial, and Courier New, respectively.
- When using other fonts than the three mentioned above for printing, the preference Print*DestinationFormat should be set to *MSWPrint* to get the best result for symbols that are adjusted to the text size.

Adjusting a File Preference



You may assign the preference parameter any file name, entered as a text string into the text field.

- You can also click the folder button, which issues a file selection dialog in which you can select a file.

Customizing the On-Line Help

In SDL Suite and TTCN Suite, you can request on-line help by selecting a menu choice in a *Help* menu, a pop-up menu or by clicking a *Help* button in a dialog. For more information about *Help* menu choices, see "Help Menu" on page 15 in chapter 1, *User Interface and Basic Operations*.

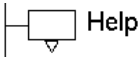
To be able to view the on-line help, you need to have one of the following HTML-viewers installed:

- Firefox
- Netscape Navigator
- Microsoft Internet Explorer

Also, you have to make sure that the on-line help files (in HTML-format) have been installed.

In the on-line help, you can also find information about how to use it.

Configuring the Help Environment



1. In the Preference Manager window, locate the icon symbolizing the help preferences.
2. Double-click the icon. This will expand it and list the available help preferences.
3. Select and, if required, adjust the preference *HelpViewer* to an adequate value, either *Firefox*, *Netscape* or *Internet Explorer*. Use the option menu that appears at the bottom of the window to select the value.
4. Select and, if required, adjust the preference parameter *FirefoxCommand*, *NetscapeCommand* or *InternetExplorerCommand* to a command that starts HelpViewer in your computer environment. Type the command in the text field that appears at the bottom of the window.

Note:

The command must be available via your PATH variable.

To limit local disk space usage in a multi-user environment, you may optionally put the on-line help files on a network server (unless you already have a network installation of SDL Suite and TTCN Suite, which is usually the case on UNIX). To do this, you move all files in the `help` directory and its subdirectories (available in the local installation directory, by default `C:\IBM\Rational\SDL_TTCN_Suite6.3`) to the network server area. This step may already have been performed on your computer system, so check where the on-line help files are stored.

Then check and adjust the preference parameter *HelpDirectory* to the help directory on the network server.

5. Select and, if required adjust the preference parameter *HelpDirectory* to the help directory (by default `C:\IBM\Rational\SDL_TTCN_Suite6.3\help` **in Windows** and `$telelogic/help/` **on UNIX**).
6. Save the preference settings and exit the Preference Manager. The on-line help is now configured in accordance to your computer environment.

Timeout Issues

When you request on-line help from SDL Suite or TTCN Suite, the Postmaster supervises the communication. On a heavily loaded computer system, the help viewer may fail to respond within the time limit specified by the environment variable `STARTTIMEOUT`. When this occurs **on UNIX**, a message is appended to the console window. The environment variable should be adjusted to a higher value to match the typical response times for the computer system where the help viewer is running.

Customizing User-Defined Preferences

Saving Parameters as User-Defined

To preserve the modifications for the future tool sessions, you must save the preference settings. The preferences are saved on a preference file denoted by `$HOME/.sdtpref` **(on UNIX)**, or `%HOME%\sdt.ini` **(in Windows)**.

Note:

In Windows: If the environment variable `HOME` is not set, the preferences are stored on a preference file in the directory from which the tool was started.

Whenever you require to save a company or project preference (and thus customize it as your personal preference), you need to confirm the operation by clicking *Save* in the following dialog:

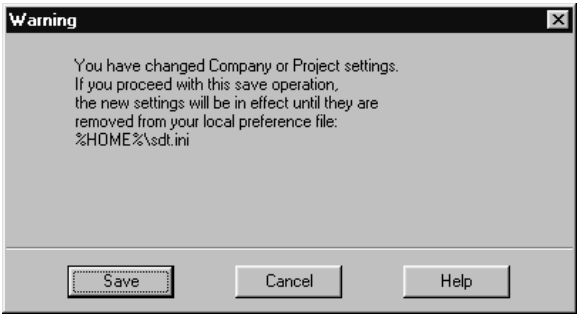


Figure 70: Confirming to modify a project or organization parameter

The reason for this is that a parameter which is defined at the company or project level should not be possible to modify inadvertently by any individual user of the tools.

Reverting User-Defined Parameters

The tools allows you to remove a preference parameter from your personal settings, in which case you will revert to the project or company preferences (or the default settings, if no project or company preferences are defined).

To revert a user-defined preference parameter, you must remove it from your preference file. The tools can do this for you:

1. Locate the preference parameter and select it.
2. From the *Edit* menu, select *Unsave*. A dialog appears, prompting you to confirm the operation.

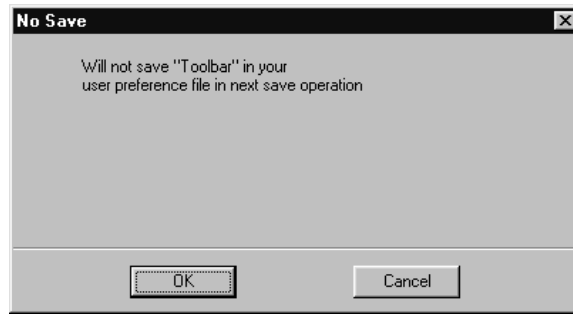


Figure 71: The Unsave dialog

– Click on the *OK* button to confirm the operation.

3. The parameter will be removed from your personal preferences next time you *Save* the preference settings.

Customizing Company Preferences

The installation may contain a file with all Preference parameters, along with their default values. The file is identified as `$telelogic/.sdtpref` **(on UNIX)**, or `sdt.ini` **(in Windows)**, and resides in the installation directory of the tools.

This file is also referred to as the *company / organization preference file*. **In Windows**, to be meaningful in a multi-user environment, the company preference file should be stored on a network drive (and, of course, all users should have read access to this file).

To customize the company's preferences, do as follows:

1. **UNIX only:** Login on a privileged account (for instance as system manager or the user who is the owner of the installation).

2. **On UNIX:** Change the default directory to `$telelogic`. Start the SDL Suite.
   In Windows: Start SDL Suite from the installation directory, i.e. **not** from a shortcut icon that defines another start directory.

3. Start the Preference Manager from the Organizer.

4. Modify the preference parameters of your choice (described in "Adjusting a Preference Parameter" on page 295). For a description of the meaning of the preference parameters, see "Preference Parameters" on page 241 in chapter 3, *The Preference Manager*.

5. Save the parameter setup. This concludes the procedure. From now on, all users (**in Windows**, those that are running from the network installation) will have the company setup as default (unless they have a different setup in their `$HOME/.sdtpref` **(on UNIX)**, or `%HOME%\sdt.ini` **(in Windows)**)

The TTCN Suite specific preferences and their default values **on UNIX** are described in "How Resources Are Read" on page 1244 in chapter 29, *Customizing the TTCN Suite (on UNIX)*.

Customizing Project Preferences

It is possible to customize the default preferences by defining a so called *project preference file*. The values defined here will supersede the company preferences.

To create a preference file that will be used by a specific project, perform the following steps:

1. **UNIX only:** Login on a privileged account (for instance as project manager or a user who has write access to a project directory).

2. **On UNIX:** Copy the company preference file to a suitable directory (where the project preference file will be stored). Change the default directory to that directory, and start the SDL Suite.
   In Windows: Copy the company preference file to a suitable directory on a network drive, where the project preference file will be stored (using a directory on a local HDD for this purpose makes no sense). Start the SDL Suite using a shortcut icon that designates this directory as the start directory.

3. Start the Preference Manager from the Organizer.

4. Modify the preference parameters of your choice (described above in "Adjusting a Preference Parameter" on page 295. For a description of the meaning of the preference parameters, see "Preference Parameters" on page 241 in chapter 3, *The Preference Manager*.

5. Save the parameter setup.

6. **On UNIX:** In a suitable file that all project members should source, insert the following statement:
   `setenv SDTPREF <directory>`

   In Windows 2000 and Windows XP, using the System properties in the Control Panel, set a `SDTPREF` environment variable to `<directory>`

   In all cases, `<directory>` denotes the project directory. Failing to do this will cause the tools to ignore the project preferences and to use company preferences instead.

7. This concludes the procedure. From now on, all project members will have the project setup as default (unless they have a different setup in their `$HOME/.sdtpref` **(on UNIX)**, or `%HOME%\sdt.ini` **(in Windows)**).

Locating the Preference Sources

To obtain information about the preference sources that are currently used (i.e. the directories and files that contain the company, project and user-defined preferences):

- Select *Info* from the *File* menu. A message box is issued:

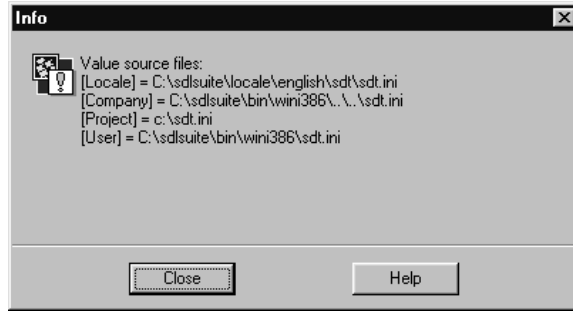


Figure 72: The message with information about preference sources

Chapter 4 Managing Preferences

Chapter

5

Printing Documents and Diagrams

This chapter describes the different *Print* dialogs in SDL Suite and TTCN Suite. It also describes the differences in printing between the Organizer, the SDL Suite and the TTCN Suite.

In the beginning of the chapter you can also find some introductory examples of how to print.

General

You can print from virtually all SDL Suite and TTCN Suite tools that provide a graphical user interface. It is possible to print the information that is managed by an individual tool, such as a graphical editor or a viewer, or to print all (or parts of) information that is related to a document structure, from the Organizer.

Print Dialogs

You specify the print options in *Print* and *Print Setup* dialog. These dialogs have different appearances, depending on where they are invoked from – the Organizer, the TTCN Suite or an SDL Suite tool. The TTCN Suite **in Windows** also provides a print preview.

Output Formats

The following output formats are provided:

- *PostScript Output* (including *Encapsulated PostScript*)
- *FrameMaker Output*
- *Interleaf Output*
- *Web Files (HTML+PNG)*
- *MSWPrint Output (Windows only)*
- *Microsoft Word Output (Windows only)*

When you print a TTCN document, only PostScript output is available. However, in the TTCN Suite **in Windows**, it is also possible to export a document to HTML, see "Converting to HTML" on page 1295 in chapter 30, *Editing TTCN Documents (in Windows)*.

Print Preferences

It is also possible to set the default print options. For more information, see "Print Preferences" on page 262 in chapter 3, *The Preference Manager*.

Printing Documents – Some Examples

This section is a brief guide to how to print some types of documents. The remaining sections of this chapter contains more detailed reference information about the print function and the *Print* dialogs.

You can print documents either from the Organizer or from within an individual tool.

Printing from the Organizer

This is an example of how to print a table of contents, one or more SDL interaction diagrams and type views from the Organizer. It is assumed that you have SDL interaction diagrams included in the Organizer and that a Type Viewer is running. The example describes how to print specific diagrams, but what is explained here may of course apply to other types of diagrams.

To print from the Organizer:

- Select *Print* | *Print All* from the *File* menu.
  - Alternatively, click the quick-button for *Print*.

  The *Print* dialog will be issued, where you may change the print settings.

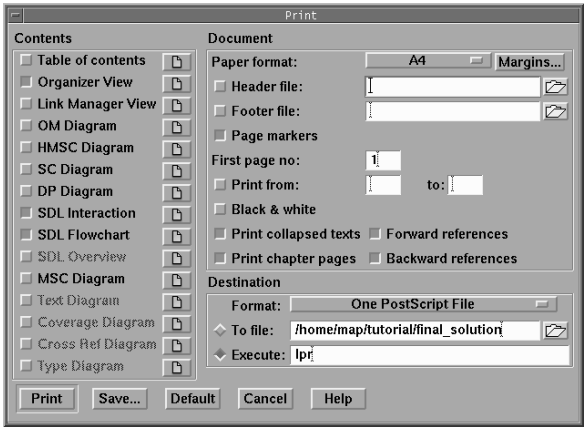


Figure 73: The Organizer Print dialog

You will start by changing the options in the *Contents* area:

1. Turn the *Table of contents* toggle button on.
2. Turn the *SDL Interaction* toggle button on.

3. Click the setup button to the right of the *SDL Interaction* toggle button.

   The *Print Setup* dialog will be issued where you may specify other options:

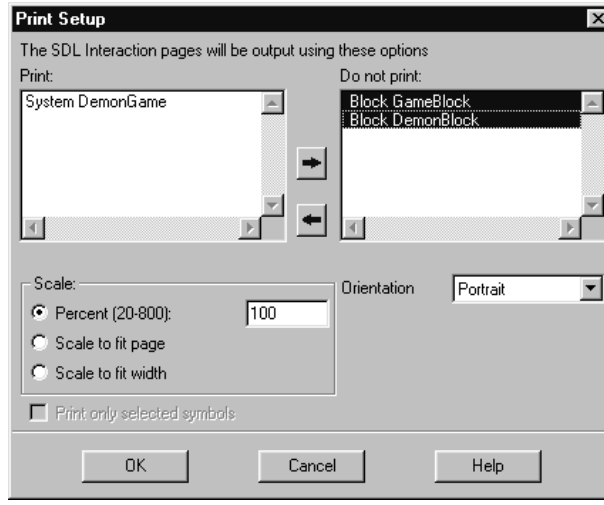


Figure 74: The Print Setup dialog

– You select what to print by using the arrow buttons to move diagrams between the *Print* and *Do not print* lists. Multiple selection is possible.

– You may change the *Scale* of the printout.

– You may change the *Orientation* of the printout.

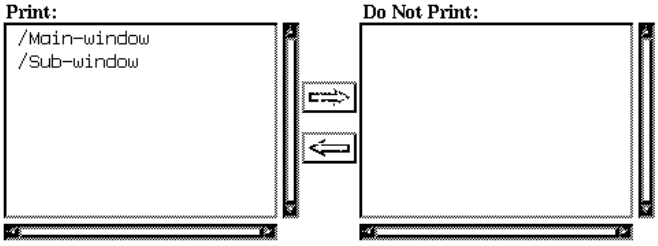
4. Click *OK*.

   The dialog will be closed.

5. Turn the *Type Diagram* toggle button on, and click the setup button to the right of the toggle button.

   The *Print Setup* dialog will be issued and you may set additional options.

   – In the *Print* and *Do not print* lists, you may specify if any of the Type Viewer main window or Tree window (identified as *Subwindow* in the list) is to be excluded.



– You may also change the other settings in the dialog.

6. Click *OK*.

   The *Print Setup* dialog will be closed.

You may also want to specify the options in the *Document* area and the *Destination* area of the *Print* dialog:

1. Select the *Paper format* that you want to use. For Word printing **(Windows only)** select the *Image format*.

2. Click the *Margins* button, to open the *Paper Format Setup* dialog.

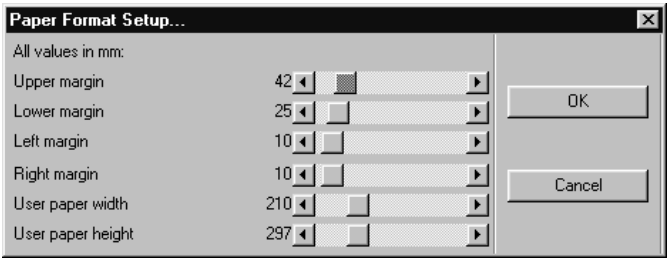


Figure 75: Setting the margins

In the dialog, you may change the paper margins, and in addition you may define your own paper size.

For Word printing **(Windows only)** there is a *Size* button instead. A click on this button allow you to define the size of the generated images imported into Word.

3. You may also want to change other settings in the *Document* area.

4. Select the output format in the *Format* option menu. You may choose between:

   – *One PostScript File*, suitable if you are going to print the documents on a PostScript printer.

   – *One EPS File* (Encapsulated PostScript), suitable if you are going to include graphics in a document. Select *One EPS File Per Page* if multiple pages are to be printed.

   – *One FrameMaker MIF File*, suitable if you want a file that can be imported into FrameMaker. Select *One MIF File Per Page* if multiple pages are to be printed.

   – *Import into FrameMaker (UNIX only)*, suitable if you want to import the file directly into a currently opened FrameMaker document.

   – *One Interleaf IAF File*, suitable if you want a file that can be imported into Interleaf. Select *One Interleaf IAF File Per Page* if multiple pages are to be printed.

   – *MSW Print* **(Windows only)**, which will make the print function use the printer you have set up with the Microsoft Windows Print Manager.

   – *One Word DOC File* **(Windows only)**, suitable if you want a file that can be used in Microsoft Word 2003 or Word 2000.

   – *Word Files (DOC + EMF)* **(Windows only)**, suitable if you want to edit the document file in Microsoft Word and be able to regenerate the pages.

   – *Web Files (HTML+PNG)*, suitable if you want to publish the document on the Web. HTML (text) and PNG (picture) files are produced that can be viewed in a web browser.

5. Select the destination of the printout – a file or a printer.

   – If you want to save the printout on a file, you should make sure that the *To file/Map file* radio button is on. The button is labelled *Map file* if multiple pages are to be printed (when you have selected a *One file Per Page* option in the format menu). The map file will contain a translation table that shows the correspondence between the input diagrams and the generated files.

– If you want to send the printout to a printer, you should ensure that the *Execute* radio button is on. You also have to type a printer command in the field, for example `lpr`. (It is possible to enter any post-processing command, such as a PostScript pre-viewer like `ghostview`.)

6. Click the *Print* button to generate the printout.

For more information on how to use advanced printing features, see "Advanced Print Facilities" on page 357.

Printing from the SDL Suite

In these examples you will learn how to print from the MSC Editor and the Type Viewer. As most settings in those print dialogs are similar to the ones in the Organizer dialog, described above, the explanations below will be quite brief.

Printing from the MSC Editor

1. Open a diagram in the MSC Editor.
2. Select *Print* from the *File* menu.

   The MSC Editor *Print* dialog will be issued.
3. Optionally, change the settings in the dialog.
4. Click the *Setup* button.

   The *Print Setup* dialog will be issued, where you may specify some additional options:

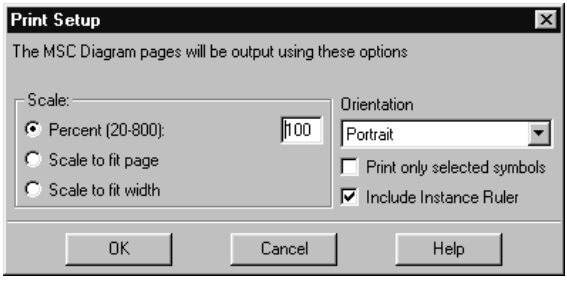


Figure 76: The MSC Editor Print Setup dialog

5. Turn the *Print only selected symbols* toggle button on, if you have selected parts of the MSC diagram that you want to print. Deselected symbols will not be printed.
6. Turn the *Include Instance Ruler* toggle button on, if you want to add an instance ruler into each printed page of an MSC. This means that the instance heads will be visible on each page – useful if you have a vertically extended MSC diagram that will cover more than one page.
7. Click the *OK* button in the *Print Setup* dialog.
8. Click the *Print* button in the *Print* dialog.

   The printout will be generated.

Printing from the Type Viewer

1. Make sure the Type Viewer is running.
2. Locate either the main window (the window labelled *Type Viewer*) or the Tree window.
3. You may also want to change the contents of the window by using commands in the *View* menu.
4. Select *Print* from the *File* menu in either the main window or the Tree window of the Type Viewer.

   If you select *Print* in the main window, the printout will be a type list. If you select *Print* in the Tree window, the printout will be an inheritance tree.

   The *Print* dialog will be issued.
5. Adjust, if required, the options in the *Print* dialog and the *Print Setup* dialog.
6. Click the *Print* button.

   The printout will be generated, based on the contents of the window that the *Print* command was selected in.

Default Scope of Print

In this section you will find information on what will be printed by default if you do not change any settings. It may also be possible to select what documents – of parts of documents – to print. See "Print Setup Dialogs" on page 337 for more information on how to include more in the printout or how to restrict it.

> **Note:**
> Link endpoints are never shown in a printed document.

Printing from the Organizer

The default printout in the Organizer depends on selections and which tools are running:

SDL, OM, SC, MSC and HMSC Diagrams; Text Documents; Cross Reference Files

If choosing Print | Print All or Print | Print Selected and no item is selected in the Organizer, **all documents** that are managed by the Organizer (i.e. are visible in the Organizer view) will by default be included in the printout. If choosing Print | Print Selected and an item is selected in the Organizer, only the document(s) included in that item, and their sub-structure (if any), will by default be included in the printout.

Coverage Files

Coverage files cannot be included in the structure that is managed by the Organizer. The graphical presentation of a coverage file is computed as you request it by running the Coverage Viewer.

Therefore, the graphical presentation of coverage information will be included in the printout if the Coverage Viewer is **running**. The resulting printout will reflect the current contents of the windows, that is, the *Main Window* and the *Coverage Details Window* of the Coverage Viewer.

SDL Types and Type Instances

These entities cannot be included in the structure that is managed by the Organizer. Furthermore, this information is computed automatically by the Type Viewer. Therefore, the Type Viewer must be running in order

to include the information in the resulting printout. The contents of the *Main Window* and *Tree Window* of the Type Viewer may be included.

TTCN Documents

If a TTCN document is selected in the Organizer, it is possible to print it, and the default scope of print is the selected document. A TTCN document cannot be included in a global printout, i.e. from the main *Print* dialog in the Organizer.

Printing from the SDL Suite

When you are going to print from an SDL Suite tool, the printout will be the information that is currently visible in the active tool window.

The *Setup* button issues the *Print Setup* dialog which is used for specifying the scope of print. For example, if you print from the SDL Editor, the default scope of print is the active SDL Editor window, i.e. the SDL page currently being edited. In the *Print Setup* dialog, the scope of print may include any of the pages contained in the SDL diagram, or be restricted to the SDL symbols currently selected in the active window.

OM, SC, HMSC, SDL, MSC and Text Editor

The default scope of print is the active Editor window, i.e. the diagram page or the MSC currently being edited.

SDL Overview Viewer

The default scope of print is the active SDL Editor window, i.e. the SDL Overview diagram currently being viewed.

Type Viewer, Index Viewer and Coverage Viewer

In a Viewer, the scope of print is the contents of the window from where you invoked print.

Preference Manager and Link Manager

The scope of print is the current contents of the window.

Printing from the TTCN Suite

The default scope of print is the entire, currently active, TTCN document. The printout will contain one table per page.

The *Print* Dialogs in the SDL Suite and in the Organizer

To print, you select any of the submenus in the *Print* menu from the *File* menu. This will open a *Print* dialog. Since the print functions in the tools are not identical, the print dialogs look somewhat different depending on where they are invoked from. As you can see in Figure 77 and Figure 78, the differences are that the Organizer print dialog includes the *Contents* area, whereas the SDL Suite tool print dialogs include a *Setup* button.

The main differences between printing from the Organizer and an SDL Suite tool are:

- In an SDL Suite tool, it is only possible to print the information or document that is opened in the tool. In the Organizer, you can select to print all or any of the documents that are managed by the Organizer.
- It is possible to print only selected symbols (when applicable) from an SDL Suite Editor, which is not possible from the Organizer.
- When you print from a viewer, the printout will be the contents of the active window. If you print from the Organizer, it is possible to include or exclude a window from the printout, independently of which window is currently active.
- In the Organizer it is also possible to print a table of contents and the contents of the Organizer window.

Note:

Endpoints are not displayed in printed documents/diagrams, regardless of the value of the *Show Link Endpoints* option.

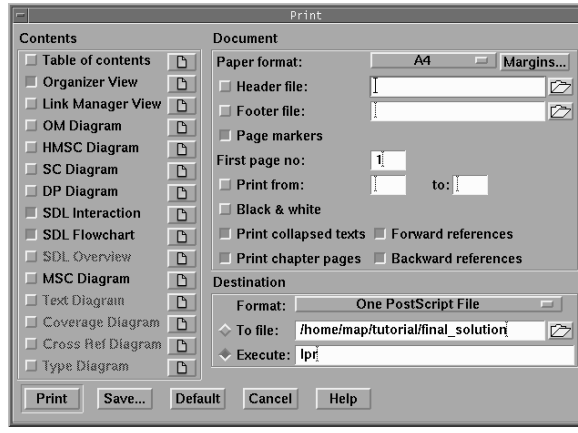


Figure 77: The Organizer Print dialog

Figure 78: An SDL Suite tool print dialog

Both dialogs contain the following areas and buttons:

- *Document Area*
- *Destination Area*
- *Print* button
- *Default* button
- *Cancel* button

The Organizer print dialog also contains the *Contents* area with setup buttons, and a *Save* button. A *Setup* button is also included in each the SDL Suite tool print dialog. The setup buttons in the *Contents* area of the Organizer print dialog are equivalent to the *Setup* button in an SDL Suite tool print dialog. The *Save* button is used to save print dialog settings in print selection files, for later use in the print dialog.

All dialog areas and buttons will be described in this chapter.

Contents Area in the Organizer Dialog

The *Contents* area is only available in the Organizer *Print* dialog. In the area, it is possible to control what information to print, through a number of toggle buttons and *Setup* buttons. Each of the toggles identify a group of documents that may be printed:

- Table of contents
- Organizer view
- Link Manager view
- OM diagram
- HMSC diagram
- SC diagram
- SDL interaction
- SDL flowchart
- SDL overview
- MSC diagram
- Text document
- Coverage diagram
- Cross reference diagram
- Type diagram

Note:

It is not possible to include TTCN documents in a global printout from the Organizer. See "The Print TTCN Dialog in the Organizer" on page 331 for information on how to print TTCN documents from the Organizer.

Some toggle buttons may be turned on by default. This is determined by which document types that are present in the default scope of print.

Table of Contents

This option determines whether a table of contents should be generated or not. The table of contents consists of a textual list with information about what source diagrams and generated diagrams that are included in the printout, with references to physical page numbers. There is also a possibility to only include Organizer chapters in the table of contents, i.e. all other printed entities are suppressed in the table of contents.

The table of contents is either printed on the first pages that constitute the resulting printout, or after an initial text document acting as a title page.

Organizer View

This option determines whether a printout of the *Organizer main window* should be included or not in the generated output. Only the visible parts (i.e. expanded nodes) are included.

The resulting printout will match the Organizer's *View Options*, i.e. file names, directories, etc. will be shown if they are in the Organizer Main window.

Link Manager View

This option determines whether a printout of the *Link Manager Window* should be included or not in the generated output. Only the visible parts (i.e. expanded nodes) are included.

OM/HMSC/SC/MSC/Text Diagram, SDL Interaction, SDL Flowchart, SDL Overview

These options determines whether the documents and diagrams that are visible in the Organizer should be included or not in the printout. The SDL *Interaction diagrams are: system, system type, block, block type, substructure, package diagrams.* The SDL *Flow diagrams are: process, process type, service, service type, procedure, operator, macro diagrams.*

Note:

Endpoint markers will not be included in printed diagrams.

Coverage Diagram

This option determines whether *Transition Coverage* and *Symbol Coverage* trees should be included or not in the generated output.

Cross Ref Diagram

This option determines whether *Definitions and Uses* should be included or not in the generated output.

The resulting output will show the graphical appearance as when displayed in the *Index Viewer*, with the exception that only SDL entities defined or used in SDL diagrams printed together with the index will be visible in the index.

Type Diagram

This option determines whether an SDL-92 Type list and SDL Type Inheritance and Redefinition list should be included in the generated output or not.

The resulting output will show the SDL-92 type lists for the SDL system currently in view in the Organizer. The lists will be produced by using the options defined in the *List Options*, *Tree Options* and *Symbol Options* of the Type Viewer.

Setup Buttons in the *Contents* Area

Furthermore, each group of documents that may be printed is supplied with setup buttons. When you click a setup button, a *Print Setup* dialog will be issued, see "Print Setup Dialogs" on page 337. In the dialog, it is possible to set additional options that affect the printout of the current group of documents.

Document Area

The *Document* area contains a number of settings that make it possible to specify the size of the paper to use, the print range and additional information to be printed on each individual page.

The *Document* options are:

- *Paper format*
- *Image format (Windows Only)*
- *Margins*
- *Header file*
- *Footer file*
- *Page markers*
- *First page no*
- *Print from/to*
- *Black & white*
- *Print collapsed texts*
- *Print chapter pages*
- *Forward references*
- *Backward references*

Note:

When using a printing format other than One Postscript File or MSWPrint the only enabled options in the Document area are Black & white (not for Frame or Interleaf printing) and Print collapsed texts. For these formats the paper layout is determined by the special Print preferences starting with *Frame*PaperFormat*. However when using Word format the margins and layout are determined by the defaults given by the created Word document.

Paper format

The *Paper format* option menu specifies what paper format the print function will use. The paper formats are:

- *A4*, the European standard size(210 * 297 mm)
- *A3*, the European standard double size (297 * 420 mm)
- *US Letter*, the American standard (8.5” * 11”)
- *US Legal* (8.5” * 14”)
- *User Defined*, which makes it possible to customize a size with the *Margins* button. The size may also have been set in the *Preferences Manager*.

Margins

The *Margins* button provides access to the *Paper Format Setup* dialog where the print margins may be specified.

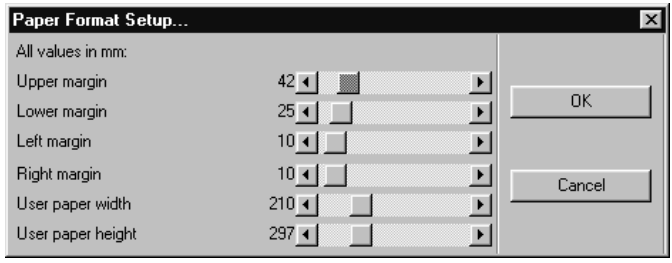


Figure 79: The Paper Format Setup dialog

The print margins govern how much space is reserved around the area used when printing. Values are expressed in millimeters.

- There are four slide bars for the adjustment of the margins:
  - *Upper margin*
  - *Lower margin*
  - *Left margin*
  - *Right margin*

  Margins are calculated with respect to the closest border and remain unaffected when rescaling, changing the paper size or the orientation of the printout.
- The dialog also contains two slide bars used for controlling what paper format to use when the *Paper format* option *User Defined* is set:
  - *User paper width*
  - *User paper height*
- The *OK* button applies the values as currently defined and closes the dialog.
- The *Cancel* button closes the dialog without changing any values.

Image format (Windows Only)

For Word printing the *Image format* option menu specifies what image format the print function will use. The image formats are:

- *Normal.dot*

  The image size will be adjusted to fit in the size calculated from a newly created Word document.

- *User Defined size*

  Makes it possible to customize an image size with the *Size* button. The size may also have been set in the *Preferences Manager*.

Header file
Footer file

This feature controls whether or not a header/footer should be printed on each page. The page header/footer is defined in a text file of its own, which you need to supply. For information about the contents and syntax of this file, see "Footer and Header Files" on page 344.

- The header/footer file options are turned on/off with toggle buttons.
- When you click the folder button, a *Select File* dialog will be issued. In this dialog you may select the header/footer file.
- It is also possible to type the name and directory of the file directly into the text field.

> **Note:**
>
> Header or footer files are only supported in *PostScript Output* (not EPS) or *MSWPrint Output (Windows only)*.

Page markers

An SDL/OM page or an MSC diagram may require multiple physical pages when printed. If you want to print an SDL/MSC/OM document and the document is physically spread over more than one physical page, the *Page markers* toggle button facilitates the reassembling of the paper sheets into the original page.

- If the toggle button is on, adjacent page markers will be inserted on the edge of each physical page. An adjacent page marker looks like a small arrowhead which refers to the adjacent page. If there are lines crossing a physical page border, line identifiers (like X1, X2... or Y1, Y2...) will be attached to the line, making it easy to find the continuation of the line on the other page.

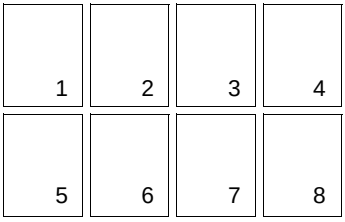


Figure 80: The page numbering

Physical page numbering follows a "first right, then down" fashion.

Note:

The physical page breaks are indicated with dashed lines in the diagram editors. A page number is inserted at the lower right corner of each page in the editor.

Page markers and all other options where page numbers are used are only supported in *PostScript Output* (not EPS) or *MSWPrint Output (Windows only)*.

It is possible to print an individual page number on each printed page. This is done by using a header file. Read more about header files in "Footer and Header Files" on page 344.

First page no

When you order multiple print jobs, you may want to restart the page numbering on each printout with a number different from 1, which is the default value.

If you want to change the number of the first page, you should type the number in the *First page no:* field.

Print from/to

It is possible to exclude a number of pages form a printout. To specify the print range, you type the page numbers in the *Print from* and *Print to* fields. Make sure that the toggle button is on, your settings will have no effect otherwise.

When the toggle button is off, all physical pages that constitute the printout will be generated.

Note:

The print range must be in accordance with the page numbering; i.e. the offset specified in the *First page no* should be added.

The physical page breaks are indicated with dashed lines in the diagram editors. These page breaks are valid only if the printing scale is not changed.

Black & white

If this option is selected, all symbols and lines will be printed with a black border on a white background. If this option is not selected, symbols and lines will be printed with the color they have on screen. As default, this option is off.

For FrameMaker or Interleaf printing this option is always on and disabled as the printing will always be in black and white.

Print collapsed texts

Text symbols, comment symbols and text extension symbols can be collapsed. When they are collapsed, it is not possible to see the complete text these symbols contain in the diagram. Instead, it is possible to print the complete text as a separate text page after the diagram. This option decides if such a text page should be printed or not. As default, this option is on.

Print chapter pages

(Organizer only.) Decides if text connected to chapter symbols in the Organizer should be printed or not. As default, this option is on.

Forward references

(Organizer only.) Decides if forward paper page references should be printed or not. Paper page references makes the printout easier to read. You can get paper page references for the following relations:

- SDL reference symbol -> SDL diagram
- (H)MSC reference symbol -> (H)MSC diagram
- SDL join/out-connector -> SDL label/in-connector
- SDL procedure call -> SDL procedure diagram
- SDL nextstate symbol -> SDL state symbol

Note:

The three last relations will only be printed if there exists an index/cross reference file in the Organizer view with up-to-date information about the SDL system. You can get an index for a correct SDL system by pressing the *Generate Index* Organizer quick button before bringing up the print dialog

As default, forward paper page references are printed.

There is a preference, Print*MaxPageReferences, that decides the maximum number of page references in one place. The default value is 20. A value of 0 is the same as “no limit”.

Backward references

Similar to forward references, but produces paper page references that allows the reader of the printout to follow the flow backwards instead.

Destination Area

The *Destination* area contains print options that affect the output format and the destination of the resulting output. It is for example possible to send a printout to a printer or to look at the results in a pre-viewer.

The options are:

- *Format*
- *To file / Map file*
- *Execute*

Format

This feature controls what output format will be generated when printing. You select the output format in an option menu. The output formats supported are:

- *One PostScript File*

  This option produces one self contained PostScript document.

- *One EPS File*[1]

1. EPS stands for Encapsulated PostScript

If only one page is to be printed, this option is valid and will result in an EPS File containing one physical page. If the *Print setup* caused more than one physical page to be printed, only the first page will be printed. The layout of the page for EPS printing is determined by the preferences in *Frame*PaperFormat*.

- *One EPS File Per Page*

  This format implies that the output will be in the form of multiple EPS files, placed in a specified directory. Along with the EPS files, a *map file* containing the translation scheme is produced. See "To file / Map file" on page 329.

- *One FrameMaker MIF File*[1]

  This format signifies the generation of one FrameMaker MIF file. The file will contain a number of contiguous, cropped anchored frames. The layout of the page for MIF printing is determined by the preferences in *Frame*PaperFormat*.

- *One FrameMaker MIF File Per Page*

  This option produces one FrameMaker MIF file per page. A *map file* is also produced, showing the translation table. See "To file / Map file" on page 329, below.

- *Import into FrameMaker* **(UNIX only)**

  If an instance of FrameMaker is found up and running, this option generates a FrameMaker MIF temporary file that will be imported into an anchored frame, placed below the current insertion point in the active FrameMaker document. If no instance of FrameMaker is found, then an attempt is made to start one with the command `maker` and create a new FrameMaker document, showing the contents of the file.

1. MIF stands for Maker Interchange Format.

Note:

To have this print option function properly, the SDL Suite and FrameMaker must run on the same computer.

Temporary files are stored on the directory designated by the environment variable `TMPDIR`. You have to set up the variable as a complete path specification such as **`setenv TMPDIR /tmp`**
To specify `TMPDIR` such as **`setenv TMPDIR .`** is not sufficient.

See also "Importing into FrameMaker (UNIX only)" on page 353 for how this option functions when multiple instances of FrameMaker are running on your computer.

- *One Interleaf IAF File*[1]

  This format produces one Interleaf ASCII Format file. The layout of the page for IAF printing is determined by the preferences in *Frame*PaperFormat*.

- *One Interleaf IAF File Per Page*

  This option produces one Interleaf ASCII Format file per page. A *map file* is also produced, showing the translation table. See "To file / Map file" on page 329.

- *MSW Print*

  (Windows only)

  The Print function uses the printer that has been set up with the Microsoft Windows Print Manager. If a printer that does not support PostScript is used, this option makes sure the printer driver available in Microsoft Windows is used.

- *One Word DOC File*

  (Windows only)

  This option produces one Word Document format file for use with Microsoft Word 2003 or Word 2000. The generated image sizes can be set by the *Image format* option. When choosing *Normal.dot* the layout of the page for Word printing is determined by the margins and paper format as defined when a new document in Word is created. When choosing *User Defined* the sizes can be set to any desired size, however when they are inserted into the Word document

1. IAF stands for Interleaf ASCII Format.

if necessary they are scaled to fit inside the margins of the document.

- *Word Files (DOC + EMF)*

  (Windows only)

  This option produces one Word Document format file that is a hub for a set of Enhanced Metafiles. When you select this option you also have to specify the main Word Document file in the *To file* text field. The produced Enhanced Metafiles are auto-named and placed in the same directory as the main Word Document file. The generated image sizes can be set as for *One Word DOC File.*

- *Web Files (HTML+PNG)*

  This option produces HTML (text) and PNG (picture) files that can be viewed in a web browser. The PNG file format is similar to the GIF file format and is supported by Netscape Navigator 4.0 and Microsoft Internet Explorer 4.0 or later. When you select this option, you also have to specify the main HTML file in the *To file* text field. All other produced files are auto-named and placed in the same directory as the main HTML file.

To file / Map file

If this radio button is turned on, the output will be a file. The name of this radio button depends on the format specified in the *Format* option menu. If the format selected will generate one output file – i.e. the format name ends with *Per Page* – the name will be *Map file*. Otherwise it will be *To file.*

A map file contains a translation scheme of all files generated (containing information about SDL diagram / page and the corresponding printout file). The naming algorithm of the generated files will ensure a fixed mapping between a diagram and the generated files between two subsequent Print jobs. However, if the input to the Print function is changed (its size!), this is not necessarily true.

See "Map File" on page 350 if you want detailed information on the syntax of a map file.

- When you click the folder button, a *Select File* dialog will be issued. In this dialog you may select the name of the file and where it is to be saved.

- It is also possible to type the name of the file directly into the text field. The directory where the file will be saved is the most currently selected.

Execute

If this radio button is on, the output file will be used as input to an OS command. An example of this is to load a printer with a PostScript file. It may also be used to send the resulting PostScript code to a PostScript previewer. For instance, if type a previewer command in the field – `ghostview` for example – you may preview the PostScript file.

The command `lpr` (or any other related) will send the printout to a printer.

The default command is specified in the *Preference Manager*. In the **Windows** version, the command `lpr` is available, which functions much like the corresponding UNIX print spool command.

Setup

When you click the *Setup* button (only available in an SDL Suite tool *Print* dialog), a *Print Setup* dialog will be issued. In the dialog, it is possible to set additional options. The additional options differ between the the SDL Suite tools. See "Print Setup Dialogs" on page 337 for more information.

The *Print TTCN* Dialog in the Organizer

To print one or more TTCN documents from the Organizer, you select one of the TTCN document in the Organizer and then you select *Print* from the Organizer *File* menu. This will issue the Organizer *Print TTCN* dialog.

When you print a TTCN document, the main difference between printing from the Organizer and printing from the TTCN Suite is:

- If you have more than one TTCN document included in the Organizer, you may select which of the documents to print. In the TTCN Suite, it is only possible to print a document that is displayed in the active Browser.

See also "The Print Dialogs in the TTCN Suite" on page 333.

The Organizer *Print TTCN* dialog contains options for the print range and first page number and whether the document is to be printed to a file or a printer. More options are available in the *Print TTCN Setup* dialog.

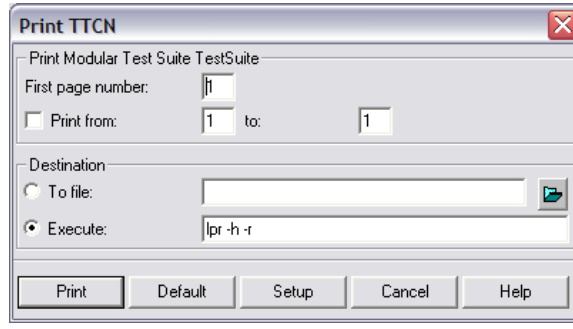


Figure 81 The Organizer Print TTCN dialog

First page number

The number that you type in this field, will be the page number inserted on the first page.

Print from/to

It is possible to exclude a number of pages form a printout. To specify the print range, you type the page numbers in the *Print from* and *Print to* fields. Make sure that the toggle button is on, your settings will have no effect otherwise.

When the toggle button is off, all physical pages that constitute the printout will be generated.

Destination Area

The options in the *Destination* area affects the destination of the printout.

To file

If this radio button is on, the printout will be a PostScript file. You also have to specify the name of the file and where it is to be saved.

- When you click the folder button, a *Select File* dialog will be issued. In this dialog you may select the name of the file and where it is to be saved.
- It is also possible to type the name of the file directly into the text field. The directory where the file will be saved is the most currently selected.

Execute

If this radio button is on, the output file will be used as input to an OS command. An example of this is to load a printer with a PostScript file. It may also be to send the resulting PostScript code to a PostScript previewer. For instance, if type a previewer command in the field – `ghostview` for example – you may preview the PostScript file.

The command `lpr` (or any other related) will send the printout to a printer.

Setup

When you click the *Setup* button, a *Print Setup* dialog will be issued. In the dialog, it is possible to set additional options, such as which of the existing TTCN documents to print. See "Print Setup Dialogs" on page 337 for more information.

The Print Dialogs in the TTCN Suite

To print from the TTCN Suite:

- Select *Print* from the *File* menu. (**On UNIX**, this menu choice is included in the Browser.)

  – **In Windows**, if the test suite overview has not been generated previously, you will have to confirm the generation before the test suite can be printed.

  In the *Print* dialog that will be opened, you may specify the page range and if the file is to be printed to a file or a printer. The output will be in TTCN-GR format – the graphical format of TTCN – according to ISO/IEC 9646-3, and it will include document indices and automatic page numbers.

When you print a TTCN document, the main difference between printing from the Organizer and printing from the TTCN Suite is:

- If you have more than one TTCN document included in the Organizer, you may select which of the documents to print. In the TTCN Suite, it is only possible to print a document that is currently displayed in the active Browser.

See also "The Print TTCN Dialog in the Organizer" on page 331.

Printing in the TTCN Suite in Windows

In Windows, the print dialogs are Windows standard. Besides the actual *Print* dialog there are also *Print Setup* and *Print Preview* dialogs.

To open a *Print Setup* dialog:

- Select *Print Setup* from the *File* menu.

  In the dialog, it is possible to specify what printer, paper size and paper orientation to use.

To open a print preview dialog:

- Select *Print Preview* from the *File* menu.

  In the dialog, the TTCN document is displayed as it will be printed and it is possible to zoom and browse the document.

Printing in the TTCN Suite on UNIX

On UNIX, you can select which Browser parts and items, for example a single table or sets of tables, to include in the printout. The selections may be arbitrary – they will be printed as a coherent document anyway. It is also possible to change the page numbering of the printout.

The settings in the print dialog will be described below:

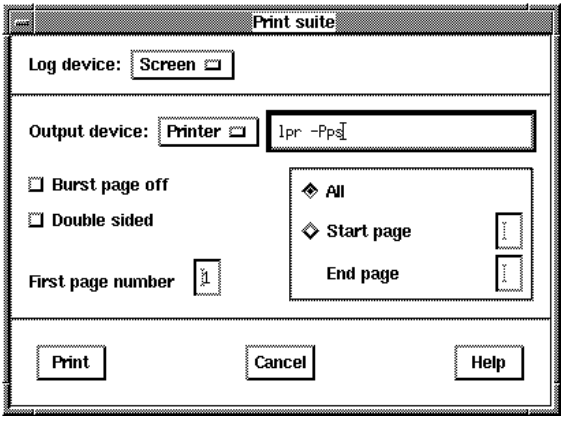


Figure 82: The TTCN Suite Print dialog **on UNIX**

- *Log device*

  Controls the log device for the print function. The default value is *Screen*, but the log can be directed to a named file (*File*) or turned off altogether (*None*).

  For more information, see "The TTCN Suite Logs" on page 6 in chapter 1, *User Interface and Basic Operations*.

- *Output device*

  If you specify the output device to be a printer, you also have to fill in a print command. If you want to print to a file, you will have to specify the destination. See also "Printing PostScript Files" on page 335.

- *Burst page off*

  If this option is set, the burst page will not be printed. The burst page is a generated, unnumbered cover page for the TTCN document, containing the print date and time.

- *Double sided*

  If this option is set, the TTCN document will be printed in a page format suitable for a double-sided copying, otherwise it will be printed in a single-sided format. The appearance of both formats is controlled by resources and can therefore be changed (see the file `<Itex.sample>` in the TTCN Suite directory).

- *First page number*

  You may change the page number of the first page. However, to ensure consistency between the page numbering of the printed TTCN document and the page numbers that appear in the TTCN document overview, make sure that:

  – The Overview Part is generated before you print.

  – The same first page number is used both in the document overview and the printout.

- Printing page ranges

  – *All* causes all pages of the document to be printed.

  – *Start page* and *End page* restricts the printout.

Example 16 ----------

If a TTCN document contains ten pages numbered 1 to 10, then *Start page* = 3 and *End page* = 3 will cause only page 3 to be printed.

Printing PostScript Files

The editable field of the *Print* dialog not only allows to change the name of the printer or to apply flags to the print command. It is also possible to insert the name of a filter for the produced PostScript code. In the public domain there exist a package of programs called `pstools` that make various magic operations with a PostScript file. The following example will print four TTCN Suite pages on each A4 page (the syntax of `pstops` is somewhat hard to read though):

Example 17 ----------

```
pstops '4:0@0.5(0.8cm,14cm)+1@0.5(10.5cm,14cm)\
+2@0.5(0.8cm,0cm)+3@0.5(10.5cm,0cm)' | lpr
```

Other filters are able to print double sided intended to bind into a book, print odd pages first etc.

By using a PostScript viewer as Ghostview instead of the normal print command, the printout of the TTCN document can be viewed on screen rather than being printed:

Example 18 ----------

```
ghostview -
```

The text in this field is controlled by the resource `Itex.print.commandPrefix`. This is by default `lpr -P` or `lp -s` depending on platform but can be altered (note that the content of the environment variable PRINTER/LPDEST is concatenated to this string). See chapter 29, *Customizing the TTCN Suite (on UNIX)* which contains a description on how to modify resources in the TTCN Suite.

Print Setup Dialogs

A *Print Setup* dialog will be issued when you click the *Setup* button in an SDL Suite tool *Print* dialog, or when you click any of the setup buttons in the *Contents* area of the Organizer *Print* dialog. When you click the *Setup* button in the Organizer *Print TTCN* dialog, the *Print TTCN Setup* dialog will be issued. The settings in a setup dialog will affect the current group of documents.

The setup dialogs look somewhat different depending on the tool they are invoked from. There are four types of print setup dialogs:

- *Print Setup* dialog for Table of contents in the Organizer, see Figure 83.
- *Print Setup* dialog in the SDL, OM, SC and HMSC Editors and, when applicable, in the Organizer, see Figure 84.
- *Print Setup* dialog in the MSC Editor, see Figure 85.
- *Print Setup* dialog in tools not mentioned above and, when applicable, in the Organizer, see Figure 86.
- *Print TTCN Setup* dialog in the Organizer, see Figure 87.

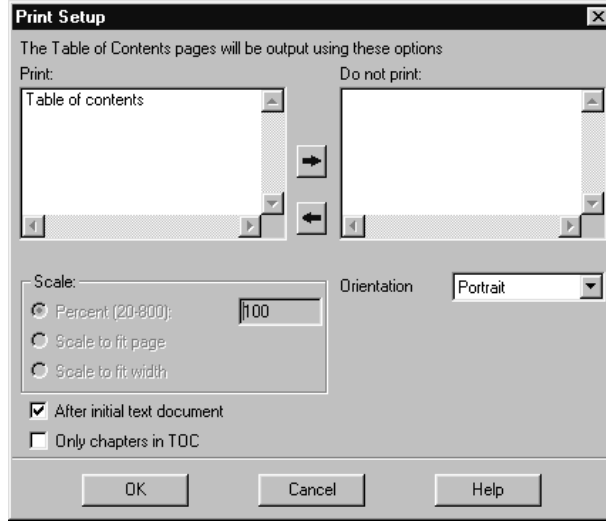


Figure 83: The Table of contents Print Setup dialog

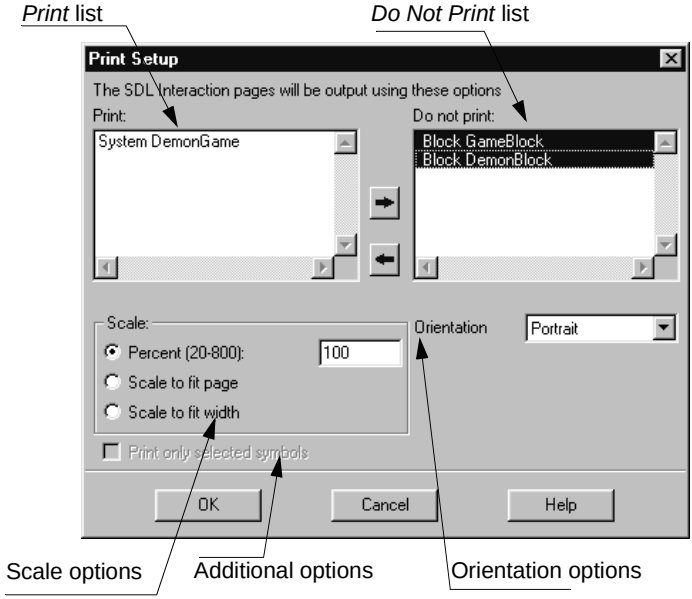


Figure 84: The Organizer Print Setup dialog

The Print Setup dialog looks the same in the Organizer and the SDL, OM, SC and HMSC Editors.

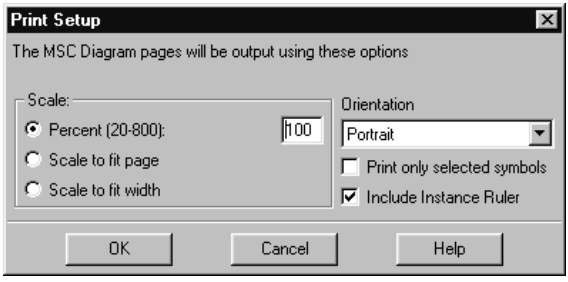


Figure 85: The MSC Editor Print Setup dialog

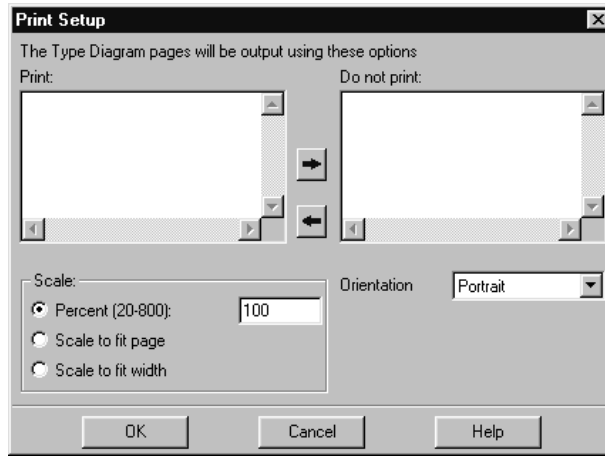


Figure 86: A viewer Print Setup dialog

The Print Setup dialog looks the same in the Link Manager, Type Viewer, the Index Viewer, the Coverage Viewer and the Text Editor.

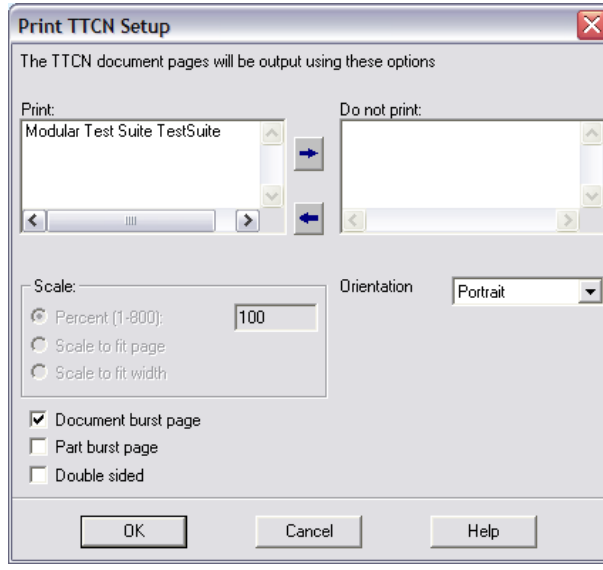


Figure 87 The Organizer Print TTCN Setup dialog

The setup dialogs contain the following:

- All dialogs contain the *Scale* area, the *Orientation* option menu, and the *OK*, *Cancel* and *Help* buttons.
- When applicable, the SDL, HMSC, OM and SC Editor and the Organizer *Print Setup* dialogs, as well as the Organizer *Print TTCN Setup* dialog, also contain the *Print / Do not print* lists.

  The lists are only included in the *Print Setup* dialog of an editor if the diagram that is to be printed consists of more than one page.

  In the Organizer, the *Print Setup* dialog will not contain the lists if you clicked the *Link Manager View* setup button.
- The *Print only selected symbols* toggle button is included in the SDL, HMSC, OM and SC Editor *Print Setup* dialogs.
- The MSC *Print Setup* dialog contains the *Include Instance Ruler* toggle button. This button is also included when MSC setup is invoked from the Organizer *Print* dialog.
- The Organizer View *Print Setup* dialog contains the *Paper page references* toggle button.
- The Table of contents *Print Setup* dialog contains the toggle buttons *After initial text document* and *Only chapters in TOC*.
- The *Print TTCN Setup* dialog contains the toggle buttons *Document burst page*, *Part burst page* and *Double sided*.

All areas and options mentioned will be explained below.

Print / Do not print

- The *Print* list shows the documents/pages that will be printed.

  If the *Print Setup* dialog is invoked from the Organizer, the list contains the documents included in the Organizer scope of print. When the dialog is invoked from the SDL, OM, SC or HMSC Editor, the list contains the current page.

  In the *Print TTCN Setup* dialog, this list shows the TTCN document that was selected in the Organizer (or all TTCN documents within a TTCN Module). Any other TTCN documents are listed in the *Do Not Print* list.

- The *Do not print* list shows the documents/pages that will not be printed.

  When invoked from the SDL, OM, SC or HMSC Editor, the list contains the pages that are included in a diagram, except the current page.

To specify what is to be printed and not:

1. Select one or several items in one of the lists, if you want to move them to the other. Multiple selection is possible.

2. Use the arrow buttons to move selected items from one list to the other.

> **Note:**
>
> **On UNIX**, you may transfer **all** diagrams/pages between the two lists by pressing `<Shift>` while clicking an arrow button.

Scale

It is possible to set the scale in almost all *Print Setup* dialogs. You can select *Percent*, *Scale to fit page* or *Scale to fit width*.

> **Note:**
>
> It is not possible to set the scale in the Text Editor *Print Setup* dialog or in the Organizer *Print TTCN Setup* dialog.

Percent

With the *Percent* radio button on, the printout of the current document category will be scaled according to the value defined in the *Percent* text field. The default value is 100% but you may of course change this.

Scale to fit page

With the *Scale to fit page* radio button on, the printout will be rescaled – if needed – to fit the physical paper size.

Scale to fit width

With the *Scale to fit width* button on, the documents will be rescaled in order to fit the physical paper width. The result becomes a column of pages constituting each document.

Orientation

In the *Orientation* options menu you may choose between the following orientations of the printout:

- *Portrait*
- *Landscape*

By changing the options, you can rotate the printout in order to optimize paper use.

> **Note:**
>
> The Orientation option is only enabled for the output formats One PostScript File or MSWPrint. Other formats use the portrait orientation but the layout can be adjusted by the preferences Print*Frame*XXX.

Additional Options

Below the *Orientation* option, there are a number of additional options, depending on which *Print Setup* dialog you have opened.

Print only selected symbols

If this toggle button is on, only objects that are selected in a window will be printed. This option is not available when you print from the Organizer. In an SDL Suite tool, it is disabled if no object is selected or if more than one window contain a selection.

Include Instance Ruler

This toggle button is only available in the MSC *Print Setup* dialog. It specifies whether an instance ruler (see "Instance Ruler" on page 1724 in chapter 39, *Using Diagram Editors*) should appear on each printed page or not. The instance ruler is, when printed, given an appearance similar to the one shown in the MSC Editor window.

Paper page references

This toggle button is only available in the Organizer View *Print Setup* dialog. It specifies whether paper page references to all printed diagrams and documents should be included in the Organizer view listing.

After initial text document

This toggle button is only available in the table of contents *Print Setup* dialog. It specifies whether the table of contents should be printed after a plain text document placed first among the diagrams and documents that are to be printed. The initial text document then acts as a title page before the table of contents.

Only chapters in TOC

This toggle button is only available in the table of contents *Print Setup* dialog. It specifies whether only the chapters in the Organizer view should be included in the table of contents, or if all diagrams and documents also should be included.

Document burst page

This toggle button is only available in the Organizer *Print TTCN Setup* dialog. If the toggle is on, a document burst page will be printed. The burst page is an unnumbered cover page for the TTCN document, where the name of the test suite and the print date are printed.

Part burst page

This toggle button is only available in the Organizer *Print TTCN Setup* dialog. If the toggle is on, part burst pages will be printed. The part burst pages are unnumbered cover pages, that separate the different parts of a TTCN document. The name of the part, the test suite and the print date, are printed on each part burst page.

Double sided

This toggle button is only available in the Organizer *Print TTCN Setup* dialog. If the toggle is on, the TTCN document will be printed with a page format suitable for a double-sided copying. The appearance is controlled by resources and can therefore be changed (see the file `Itex.sample` in the TTCN Suite installation directory).

Footer and Header Files

You may specify if a header and/or footer is to be inserted on each printed page. The first thing you have to do is to define the header and footer in separate text files. The format is ASCII based and line oriented. A number of variables are available, providing additional information of the kind of diagram printed.

Text is written on a white background, which becomes visible if the text appears on top of any graphical object.

Printing Order

The priority order for writing headers and footers is as the inverse order in which the items appear in the file. That is any text, variable or graphical symbol, overwrites an already written item. (The last row in the file has the highest priority).

Syntax

The syntax used in the header and footer files is given below in a BNF notation:

```
File            ::= ( <LINE> | <EXTENSION-LINES> |
<EPSF> ) *
<LINE>          ::= <X> <Y> ( <TEXT> | <VARIABLE> )
+ \n
<X>             ::= integer, x - position in
millimeter
<Y>             ::= integer, y - position in
millimeter
<TEXT>          ::= any ASCII text - no newlines
allowed
<VARIABLE> ::= any variable as described  in
"Variables" on page 345,
<EXTENSION-LINES> ::= any of the extensions as
defined in "Extensions" on page 347.
<EPSF>          ::= EPSF <X> <Y> <FILE>
<FILE>          ::= user provided filename
containing EPS code
```

Note:

The positions are relative the **upper left corner** of the physical paper in the header and the **lower left corner** in the footer.

Variables

The following variables are supported in header and footer files:

Variable	Explanation
<date>	As set by the Print preference *date*.
<diagramname>	The name of the diagram. Ignored if the printed diagram was not one of: • SDL diagram • SDL Overview diagram • MSC.
<diagramtype>	The type of the diagram. Ignored if the printed diagram was not one of: • SDL diagram • SDL Overview diagram • MSC.
<pagename>	The logical name of current page. Ignored if the printed diagram was not one of: • SDL diagram • SDL Overview diagram.
<file>	The current file being printed. Ignored if no corresponding file exists. It could be a *New* (unsaved) file being printed or it could be a *Type View* which has no corresponding file.
<page>	The current page number.
<directory>	The directory where the current file is located. Ignored if no corresponding file exists. It could be a *New* (unsaved) file being printed or it could be a *Type View* which has no corresponding file.
<chaptername>	The current chapter name, defined by a chapter symbol in the Organizer View.
<chapternumber>	The current chapter number, defined by a chapter symbol in the Organizer View.
<totalpages>	The total number of printed pages, excluding the title page and the table of contents.

Obsolete Variables

(i.e. variables used in SDT 2.3)

Variable	Explanation
`<area>`	Prints *Work Area* or *Original Area* in plain text depending on if the diagram was found in the source directory or not. The variable is available for backwards compatibility only.

Example 19 ----------

For example, you may have a process called Myproc that you are going to print out. First you write a footer file with the following contents:

```
10 20 Telelogic AB
10 15 <date>
150 15 <diagramtype> <diagramname> <pagename>
180 20 Page <page>
```

If the date is March 1, 1999, the 15th printed page would look something like this:

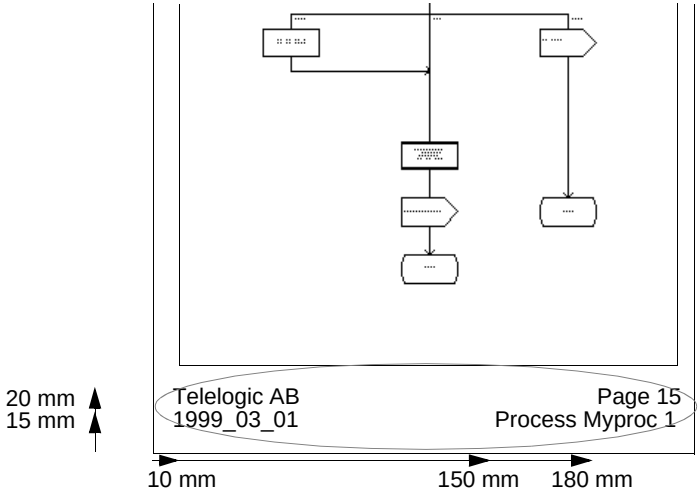


Figure 88: The resulting footer

Extensions

Below are listed the extensions that enhance the appearance of the header and footer.

Frames

The keyword `FRAME` makes a frame appear at any position on the paper. The function is intended for framing the header and/or footer.

Format

The exact format is:

```
FRAME x1 y1 x2 y2 [type]
```

where the start and stop positions are given in millimeters. The optional type argument may be either `3D` or `plain` (default).

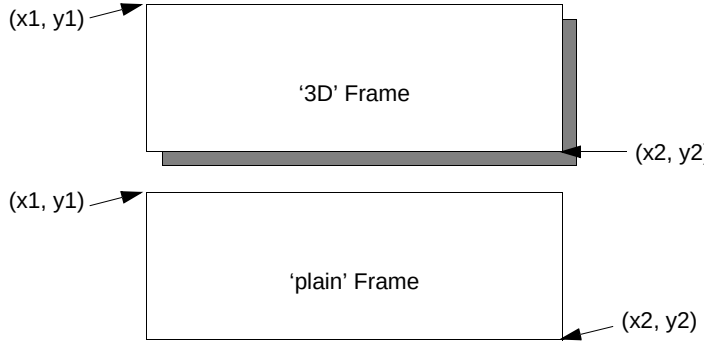


Figure 89: 3D frame and plain frame

Separators

A separator is a horizontal line which is intended to separate the header and/or footer from the data area of a printed page.

Format

The SEPARATOR statement has the general format:

```
SEPARATOR x1 y1 length [type]
```

where the start position and length of the separator are given in millimeters. The optional type argument may be either:

- Filled
- Double
- Single (default)

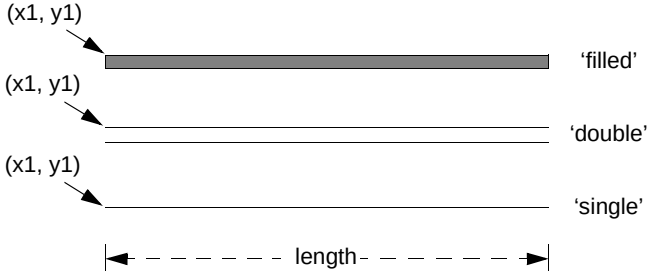


Figure 90: Filled, double and single separators

Boxes

Use the BOX keyword to make a filled rectangle appear anywhere on the paper. It can be used for highlighting sections of the header and/or footer.

Format

The format is:

```
BOX x1 y1 x2 y2 [grayscale]
```

where the start and stop positions are given in millimeters. The optional grayscale argument is a number in the range 0 to 100 (default is 50).

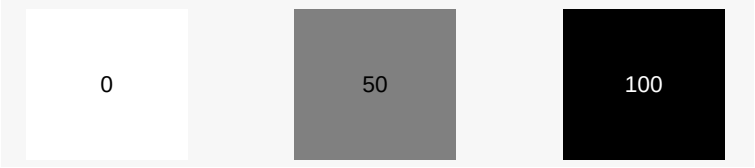


Figure 91: Boxes

Grayscale=0 = white, grayscale=50 = medium gray and grayscale=100 = black.

Defining Multiple Footers and Headers

It is possible to have a separate footer or header for the first printed page. There are three keywords that can be typed on a separate line in a footer or header definition file to achieve this:

- IFFIRST

Definitions after this keyword will only be used for the first printed page.

- IFNOTFIRST

Definitions after this keyword will be used for all pages after the first page, but not for the first page.

- ENDIF

Definitions after this keyword will be used for all pages.

Map File

When multiple output files are generated for one input diagram, a map file is also generated. It shows the translation table between input diagrams and resulting files. The name of this file should be specified in the *Map file* field.

Syntax

The map file is an ASCII text file containing lines of the format:

```
<filename> "[<diagram type>]" "[<diagram name>]"
"[<page name>]"
```

where `<filename>` is an absolute path to a generated file. The file name is composed of:

- the first four letters of the diagram name (fewer if the diagram name is shorter than four letters)
- the first two letters of the page name
- an extension.

If this naming scheme generates two files with the same name, a number (1-99) is added to the second file name, preceding the extension.

Example 20: Contents of a UNIX Map File

```
...
/usr/ope/doc/ContDe.eps "Block" "Controller"
"Declarations"
...
/usr/ope/doc/ContDe1.eps "Block" "Container"
"Declarations"
...
```

Example 21: Contents of a Windows Map File

```
...
C:\tlog\doc\ContDe.EPS "Block" "Controller"
"Declarations"
...
C:\tlog\doc\ContDe1.EPS "Block" "Container"
"Declarations"
...
```

Should either of the descriptive `<diagram type>`, `<diagram name>` or `<page name>` fields be inapplicable to an item, the corresponding field in the map file will be empty (i.e. “”).

More Information on Output Formats

PostScript Output

PostScript output may be generated as one standard PostScript file or one or more Encapsulated PostScript (EPS) files. When multiple EPS files are generated, a translation table will also be produced in a *Map File*, linking the name of a generated file to the contents.

Standard PostScript

A normal print operation generate one PostScript document. All data is represented uniformly, using the same paper format, header, footer etc. Only the scale and orientation might vary between pages.

Encapsulated PostScript (EPS)

The EPS output makes use of the scale and orientation settings for each document category. Since EPS files are not clipped, the internal scale has little importance when the file is imported into an external documentation or desktop publishing system.

Header, footer and adjacent page markers are not included in the output. Paper layout are determined from the Print preferences *Frame*Paper-Format*.

Scaling in EPS Files

The scaling options are handled as follows:

- *Scale to fit page*

  Each logical page is output as one EPS file with an internal scale adjusted to fit the entire logical page into the bounding box given by the current paper format and margins. When the EPS files are scaled into external documentation software, this will correspond to the “Fit Into Page” option for PostScript files.

- *Percent*

  As *Scale to fit page*, but with a user defined internal scale.

- *Scale to fit width*

As *Scale to fit page*, but with the internal scale adjusted to fit the width of the logical page into the space given by the paper width and the left and right margins.

Handling of Expanded Text in PostScript

Expanded texts are the text contained in such symbols that are minimized[1] in a graphical page, see "Text / Additional Heading / Package Reference Symbols" on page 1891 in chapter 43, *Using the SDL Editor*.

Standard PostScript

Expanded text is output as plain text using the font specified by the `SDT*PrintFontFamily` and `SDT*PrintTextHeight` preferences.

Encapsulated PostScript

Expanded text in EPS output is stored on separate files. One text file is generated for each logical page containing expanded text, regardless of the number of expanded texts in that logical page.

- When generating multiple EPS files, the text file for a logical page called `<x>.eps` will be `<x>.exp`.
- When printing to one EPS file, the text file for `<x>` will be `<x>.exp`.

PostScript Standard Conformance

The generated PostScript code conforms to either of these standards:

- PostScript Language Document Structuring Conventions - Version 3.0
- Encapsulated PostScript File Format - Version 3.0.

FrameMaker Output

FrameMaker output may be generated either as:

- One MIF[2] file containing *cropped anchored frames*
- One or more MIF files containing one logical page each

1. Meaning resized to a minimal size.
2. MIF stands for Maker Interchange Format.

The generated pages are the same in both cases. A translation table is also generated in the latter case.

- One temporary MIF file that can be imported directly into FrameMaker. See "Importing into FrameMaker (UNIX only)" on page 353.

Depending on the scale setting (see "Scaling in EPS Files" on page 351), each logical page will be represented as one or more *cropped anchored frames* in the output.

Header, footer and adjacent page markers are not included in the output. Paper layout are determined from the Print preferences *Frame*Paper-Format*.

Handling of Expanded Text in MIF

Expanded text (see "Handling of Expanded Text in PostScript" on page 352 for an explanation) is inserted as plain text after the anchored frames generated for the corresponding logical page. In order to preserve the appearance of the users original text, hard returns are inserted where new lines are found.

MIF Conformance

The generated MIF files conform to FrameMaker Interchange Format version 4.00, which can be read into FrameMaker version 4.x, 5.x and 6.x without problems.

Importing into FrameMaker (UNIX only)

The X Window root window has an atom (property) that governs how FrameMaker communicates with SDL Suite and TTCN Suite. The name is arbitrary, but must defined both in SDL Suite and TTCN Suite as well as in FrameMaker. In the example below, the name `_Frame_Import` will be used.

To have FrameMaker behave in accordance, you should specify what resource to be used by SDL Suite and TTCN Suite when starting FrameMaker. The resource values should be entered in the SDT resource file (in which case all users are affected) or in a suitable user X resource file. The resource is called `sdtfmimp*rpcProp`.

Example 22: X Resource file sample

```
...
sdtfmimp*rpcProp: _Frame_Import
...
```

There are multiple ways to set up FrameMaker. The resource `Maker.rpcProp` may be used for this.

Interleaf Output

Interleaf output may be produced as:

- one IAF[1] file containing *anchored frames*, or
- one or more IAF files containing one logical page each.

The generated pages are the same in both cases. A translation table is also generated in the latter case.

Depending on the scale setting (see "Scaling in EPS Files" on page 351), each logical page will be represented as one or more *anchored frames* in the output.

Header, footer and adjacent page markers are not included in the output. Paper layout are determined from the Print preferences *Frame*Paper-Format*.

Handling of Expanded Text in IAF

Expanded text (see "Handling of Expanded Text in PostScript" on page 352) is inserted as plain text after the frames generated for the corresponding logical page. In order to preserve the appearance of the users original text, hard returns are inserted where new lines are found.

IAF Conformance

The generated IAF files conform to Interleaf ASCII Format version 8.0 (used in Interleaf 5).

MSWPrint Output (Windows only)

The MSWPrint output can be used on Windows to print on any Windows printer, thus this is suitable if the printer does not support Post-

1. IAF stands for Interleaf ASCII Format.

script. As the complete print output using this option will be divided into several print jobs the printing should be done only on one side of the paper, otherwise the page numbering will not be as expected.

Handling of Expanded Text in MSWPrint

Expanded text (see "Handling of Expanded Text in PostScript" on page 352) is handled the same as for standard PostScript.

Microsoft Word Output (Windows only)

Word output can be used to print as a Microsoft Word document. A complete Word document is created and the generated pages are created as enhanced metafiles.

Header, footer and adjacent page markers are not included in the output. Paper layout are determined from the defaults when creating a new Word document. The image sizes can be set by the *Image format* option.

Handling of Expanded Text in Word output

Expanded text (see "Handling of Expanded Text in PostScript" on page 352) is inserted as plain text after the frames generated for the corresponding logical page. In order to preserve the appearance of the user's original text, hard returns are inserted where new lines are found.

Adding Printer Fonts (UNIX only)

By default only three printer fonts are available in printouts: Times, Courier and Helvetica. It is possible to add other printer fonts. The requirement is that there exist AFM (Adobe Font Metrics) files for the desired fonts. An AFM file contains the character metrics necessary for correct layout of text in printouts.

1. Locate AFM files for the regular, bold, italic and bold italic versions of the font. Many AFM files are available from Adobe Systems Inc. through on-line services (e.g. via ftp from `ftp.adobe.com`).
2. Store the files in the directory `$telelogic/fontinfo`.
3. Name the files according to the scheme in the table below. Note that you have to specify *`<basename>`* by using lower case characters (e.g. `palatino-I.afm`). Otherwise, you are free to choose any `<basename>` that does not conflict with other fonts or font files.

File contents	File name
regular font	`<basename>.afm`
bold font	`<basename>-B.afm`
italic font	`<basename>-I.afm`
bold italic font	`<basename>-BI.afm`

4. You may then choose the font as `<basename>` in the Preference Manager (case is not important in this context):

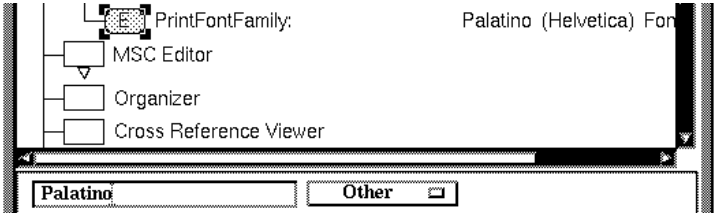


Figure 92: Specifying print font as Other.

Palatino is selected in the current example.

The generated files will use the true name of the font, extracted from the FontName and FamilyName attributes in the AFM files.

Advanced Print Facilities

This section is divided into three parts.

- The first part lists all advanced print facilities.
  See "List of Advanced Print Facilities" on page 357.
- The second part describes how to use the advanced print facilities to produce a nice printout, starting with an SDL system that is not already prepared for printing with advanced print facilities.
  See "Introducing Advanced Print Facilities" on page 368.
- The third part is about using the advanced print facilities when an SDL system is prepared for printing with advanced print facilities.
  See "Using Advanced Print Facilities" on page 382.

List of Advanced Print Facilities

The following print facilities will be described in this section:

- Headers
- Chapters
- Title Page
- State Matrices
- Index
- Fonts
- Text Indentation

Headers

You can create a textual header definition file that defines the appearance of a header that is found at the top of each printed page. All the possibilities for creating a header is described in "Footer and Header Files" on page 344, here you will only find the most important ones.

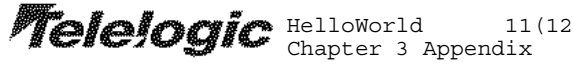


Figure 93: A header example

Creating a Header Definition File

The Organizer preference ShowHeader defines if the header symbol should be visible as default in the Organizer view. If the header symbol is not visible, you can make the symbol visible by selecting *Header File* in the *View > View Options* dialog.

Figure 94: The header symbol in the Organizer view

You can double-click on the header symbol to open a Text Editor to be able to define the header and create a header definition file. When you have defined the header (read more about some of the possibilities for that below), save the file and you are done with this step.

Defining Text in Headers

In the header definition file, insert a line like this:

```
60 20 This printout was produced by X
```

This means that 60 millimeters from the left border and 20 millimeters from the top border, the text “This printout was produced by X” will be placed.

Using Header Variables

In a header definition file, insert a line like this:

```
100 10 <page>(<totalpages>)
```

This means that 100 millimeters from the left border and 10 millimeters from the top border, a text such as “3(9)” will be placed, informing the reader of the printout that this is page three of nine.

Using Pictures in Headers

You can introduce pictures in the headers. The pictures must be in Encapsulated PostScript file format (EPSF), and they are referenced from the header definition file. If you have a picture of for instance your company logo in a file `/home/lat/logo.eps`, then you can get it into your header with the following line:

```
EPSF 10 35 /home/lat/logo.eps
```

This line means that the picture defined in the referenced file will be placed 10 millimeters from the left border, and 35 millimeters from the top border.

Separate First-page Header

You can have one header for the first page, and another header for all other pages. To achieve this, you can divide your header definition file in three parts by using three different keywords, or rather keylines:

```
IFFIRST
(Part 1. Place everything that is unique for the
first page header here.)
ENDIF

IFNOTFIRST
(Part 2. Place everything that is unique for the
other header here.)
ENDIF

(Part 3. Place everything that is common to both the
first page header and the other header here.)
```

Chapters

You can divide the diagrams and documents in your Organizer view into different chapters by using the chapter symbol. You add a chapter symbol to the Organizer view with *Edit > Add New*. In the dialog that follows, select one of the *Chapter* alternatives under the *Organizer* document type group, and specify the chapter name under *New document name*. (If you do not want to create a chapter introduction text (see below), make sure that the *Show in editor* toggle button is off before closing the dialog.)

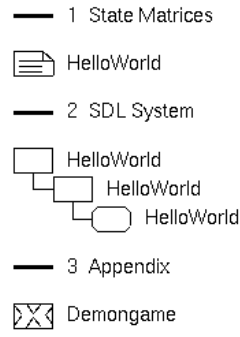


Figure 95: Chapters in the Organizer view

Chapters can be auto-numbered and can be moved with the up and down quick buttons. You can select a chapter symbol to restrict the scope for diagrams and documents that are going to be printed.

Creating a Chapter Introduction Text

It is possible to associate a text with a chapter symbol. The text will be printed at the position of the chapter symbol, and the text can be regarded as a chapter introduction text.

To create a chapter introduction text, you should invoke the Text Editor. This can be done in two ways:

- If you have not closed the *Add New* dialog, then you can bring up the Text Editor by making sure the *Show in editor* toggle button is on, before closing the dialog.
- If you have closed the *Add New* dialog, then you can double-click on the chapter symbol. In the dialog that follows, select *Edit chapter symbol*. In the second dialog that follows, make sure that the *Show in editor* toggle button is on and close the dialog to bring up the Text Editor.

In the Text Editor, type in the chapter introduction text and save the file.

Starting a New Chapter in the Middle of an SDL Diagram

For a large SDL diagram with many pages, it is convenient to divide the different pages of the diagram into different sub chapters.

To make this possible, you have to make the page symbols visible in the Organizer, if they are not visible already. This is done in the *View Options* dialog by turning on *Page Symbols*.

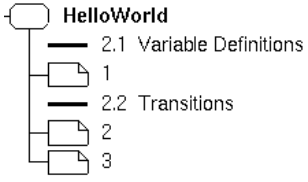


Figure 96: Chapters between SDL pages

When page symbols are visible, you can move chapter symbols into the correct place within the SDL system by using the up and down quick-

buttons. You can also create new chapter symbols directly at the correct place, by selecting the SDL symbol that the chapter symbol should be placed after, before invoking *Add New*.

Note that MSCs can be placed in the middle of an SDL system or diagram, in the same way as chapters. This is useful if you want to print an MSC close to the corresponding SDL.

Defining the Start Chapter Number

Normally, the start chapter number is 1 (or 1.1 if the first chapter symbol has the type *Chapter 1.1*.) You can change this number in the *View > Chapter Options* dialog.

Deciding Maximum Chapter Level in the Table of Contents

As default, all chapters are visible in the table of contents. If you have a lot of chapters, you can limit the number of chapters in the table of contents by using the *View > Chapter Options* dialog. A maximum chapter level of 2 will only include chapter symbols of type *Chapter*, *Chapter 1* and *Chapter 1.1* in the table of contents.

Printing a Table of Contents with Only Chapters

To print a table of contents, you have to turn on the *Table of contents* toggle button in the *Contents* section in the *Print* dialog.

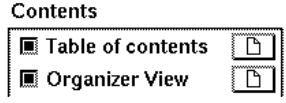


Figure 97: The Table of Contents entry in the Print dialog

As default, the table of contents includes chapters, diagrams and documents. To restrict the table of contents to just chapters, there are two ways:

- If you want to do a temporary restriction just for one printout, then there is a toggle for doing so in a sub dialog to the *Print* dialog: To get to the sub dialog from the *Print* dialog, press the setup button with a picture close to the Table of contents entry in the Contents section. In the sub dialog, turn on the toggle *Only chapters in TOC*.

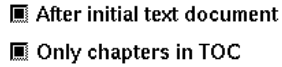


Figure 98: Part of the Print Setup dialog for Table of contents

- If you want the restriction to be more permanent, go to the *Preference Manager*. In that tool, double-click on the *Print* symbol to expand its contents, and set the value of the preference OnlyChaptersInTOC to on. Save the preferences and restart.

Title Page

A plain text document can become a title page by placing it first among the diagrams and documents that are printed. There is nothing special about a title page, except that it can be placed before the table of contents.

Figure 99: A title text document in the Organizer view

Getting the Title Page before the Table of Contents

To print a table of contents, you have to turn on the *Table of contents* toggle button in the *Contents* section in the *Print* dialog.

Normally, you do not have to do anything to get the title page before the table of contents, because the print operation will as default treat an initial text document as a title page. There is however a way to turn this on and off. This is done in the Print Setup dialog for the *Table of contents* entry in the *Contents* section in the *Print* dialog. You can get to this dialog from the *Print* dialog by pressing the setup button with a picture close to the *Table of contents* entry.

State Matrices

State Matrices are described in detail in chapter 6, *The Text Editor*. Here are only the most important things discussed, enabling you to get a nice printout of state matrices for processes in your SDL system.

Creating State Matrices

When you have a correct SDL system, you can create state matrices with the *Generate* > *State Overview* menu choice. A state information file (`*.ins`) will be created in your target directory, a plain text symbol connected to the file will be added in the Organizer view, and a Text Editor will pop up, acting as a State Matrix Viewer.

```
Process HelloWorld: nextstates
States
a Start state
b waiting
c stop
          |  STATES
SIGNALS   |  a    b    c
          |-------------
-         |  b
Hello     |       c
```

Figure 100: A state matrix

Filter State Matrices

The Text Editor has two menu choices that you can use to get the state matrices that you want.

- *View* > *State Matrix* > *Filter Processes*. This menu choice allows you to hide state matrices for SDL processes that you are not interested in.
- *View* > *State Matrix* > *Transition Information*. This menu choice decides the information that should be presented for transitions in your state matrices. As default, you will get two state matrices for each SDL process: One with paper page references to the corresponding SDL transition and one with nextstate information. You can with this menu choice also have state matrices with output/signal sending information and procedure call information.

The options in the two dialogs above can also be set in a more permanent way, using preferences. To do this, use these Text Editor preferences:

- StateMatrixFilter
- StateMatrixCalls
- StateMatrixNextstates
- StateMatrixOutputs
- StateMatrixPageNumbers

Printing State Matrices with Paper Page References

When you view state matrices on-line, you will not see any paper page references in a page number state matrix. All page numbers are undefined and replaced by a "*". The page numbers are only visible in your printout, and only if you print the SDL processes together with the state matrices from the Organizer.

Index

The index is described in detail in chapter 46, *The SDL Index Viewer*. Here are only the most important things discussed, enabling you to get a nice printout of an index of all SDL entities defined and used in your SDL system.

Creating an Index

When you have a correct SDL system, you can press the *Generate Index* quick button in the Organizer. This will pop up an index viewer with an index of all SDL entities defined and used in your SDL system.

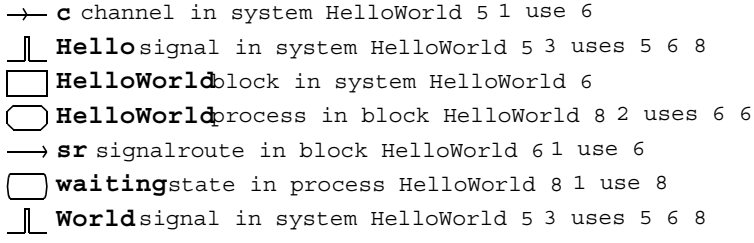


Figure 101: An SDL entity index example

Filter Index Information

The generated index may be too long to be printed in its entirety. You can make the index smaller by filtering out information that you consider unimportant.

You can filter out SDL entities defined in specific SDL diagrams. This is done with the *View > Filter Diagrams* dialog. A common use of this possibility is to filter out SDL entities defined in package Predefined.

You can filter out SDL entity types. This is done in the *View > Filter Types* dialog. For instance, if you want an index of just the signals in your SDL system, then you can use this possibility to filter out all other SDL entity types.

To make your filter more permanent than just for the current Index Viewer session, you should define your filter settings in the Preference Manager before starting the Index Viewer. These two preferences are appropriate:

- FilterDiagrams
- FilterTypes

Deciding Index Appearance

Each SDL entity in the index can be presented in four different ways. You can change the index appearance for the current Index Viewer session with the *View > Index Appearance* dialog. To make your changes more permanent, use the Preference Manager and the following preference:

- IndexAppearance

As default, the index appearance is set to *Detailed*. This setting is appropriate for on-line viewing. For a printout, it might produce unnecessary large indexes. To get a smaller index in a printout, use the *Compact* index appearance.

Printing an Index with Paper Page References

It is possible to get paper page references for both definitions and uses of SDL entities in a printout of an index. To achieve this, leave the Index Viewer window on screen while doing a printout from the Organizer that includes your SDL system and the index view.

Fonts

There are two kinds of fonts:

- Proportional fonts such as Times. These fonts have the advantage of not needing much space.
- Non-proportional fonts such as Courier. These fonts have the advantage of being useful for creating tables where it is important that the same character position on different lines are aligned.

Below, you will learn how to set up SDL Suite and TTCN Suite to use appropriate fonts for different uses.

Using a Non-Proportional Font in the Text Editor

State matrices are viewed and printed from the Text Editor. State matrices should be presented using a non-proportional font. To get a non-proportional font in printouts of state matrices, set the preference SDT*PrintFontFamily to Courier.

Using a Non-Proportional Font for SDL Text Symbols

Sometimes, it is convenient to make aligned tables in SDL text symbols. This requires a non-proportional font. On the other hand, you do not want a non-proportional font for all other symbols in a process diagram, because you want to squeeze as much text as possible into a flow symbol without going outside the symbol border.

You can have a separate, non-proportional font for SDL text symbols. This is achieved by setting Editor*FontText*TextSymbolFontFamily to Courier. This preference is valid both on-line and in print. All the other symbols use the fonts set by Editor*FontText*ScreenFontFamily and Editor*FontText*PrintFontFamily.

Title Fonts: Size and Boldness

The table of contents has a title named “Contents” that is always presented in bold.

Font size and boldness for chapter names in text page titles are determined by the chapter level:

- Chapter level 1 (for instance 1) uses 14 pt bold.
- Chapter level 2 (for instance 1.1) uses 12 pt bold.

- Chapter level 3 (for instance 1.1.1) uses 11 pt bold.
- Chapter level 4 (for instance 1.1.1.1) uses 11 pt, not bold.

A normal text, and a chapter without a chapter number (chapter level 0), uses the same font appearance for the title as the font used for the following text.

To get header titles in a larger font (14 pt) and with a bold text, use a header definition file with a *Title* keyword. (<#Title>, <#Title 2> or something similar.)

Text Indentation

There are three text indentation levels, and each text indentation level has a corresponding preference:

- Print*TextIndentation1. Used by text pages for presenting chapter numbers. Default: 44 mm from the left paper edge.
- Print*TextIndentation2. Used by text pages for presenting chapter names and normal text. Also used by the table of contents for presenting chapter numbers. Default: 67 mm from the left paper edge.
- Print*TextIndentation3. Used by the table of contents for presenting chapter names. Default: 90 mm from the left paper edge.

Introducing Advanced Print Facilities

In this sub section we will take a more detailed and practical look at how an ordinary SDL system can be turned into an SDL system with enhanced print facilities.

> **Note:**
>
> As default in this section, we are talking about menu choices and dialogs in the Organizer.

Initial Setup

Before starting doing print enhancements for the specific SDL system, we should set up some preferences affecting how the final printout will look like.

1. Bring up the Preference Manager with the menu choice *Tools* > *Preference Manager*.
   - To get a non-proportional font in printouts of state matrices, set the preference SDT*PrintFontFamily to Courier.
   - To get a non-proportional font in text symbols in SDL diagrams, set the preference (SDL) Editor*FontText*TextSymbolFontFamily to Courier.
2. Save the preferences and exit the Preference Manager.

The Example SDL System

We will use an SDL system called HelloWorld as an example. When we start, we have a correct SDL system that is not adapted in any way for producing a nice printout. If we just do a plain printout of HelloWorld, we will get the following:

> **Note:**
>
> The figures below are not exact copies of a printout, they have been modified slightly to fit in this manual.

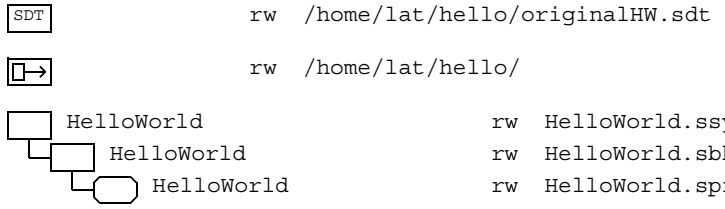


Figure 102: HelloWorld, Organizer view

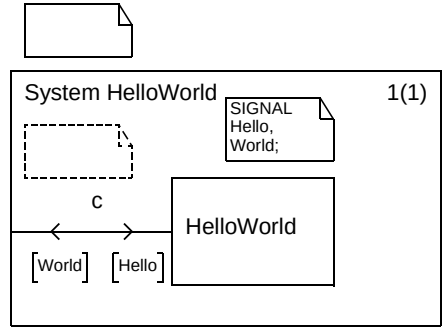


Figure 103: HelloWorld, system diagram

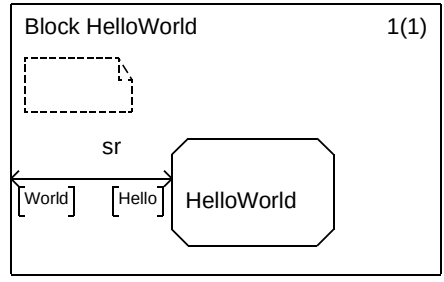


Figure 104: HelloWorld, block diagram

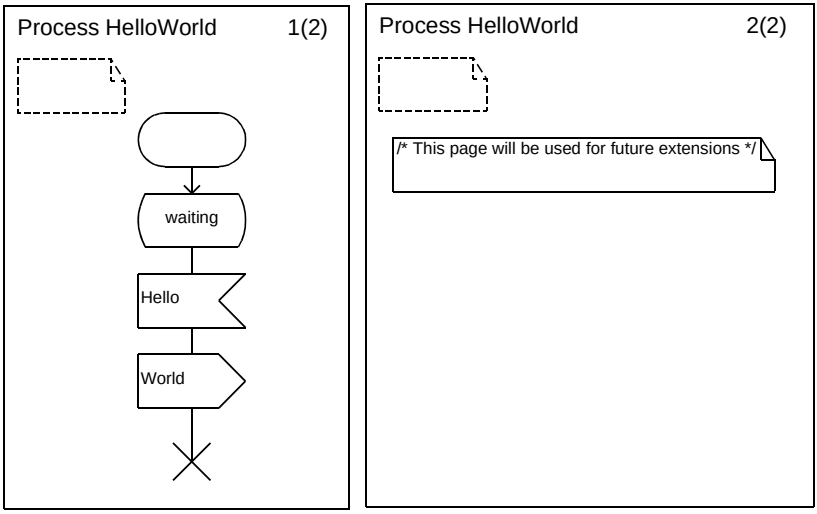


Figure 105: HelloWorld, process diagram

To get a better printout, we will add the following:

- A Header on Each Printed Page
- A Title Page
- Chapters and Sub Chapters
- State Matrices for the SDL Process
- An Index of SDL Entities

When we have done that, we will perform a print operation to actually get our enhanced printout.

A Header on Each Printed Page

The header should have the following properties:

- A separate header appearance for the first page.
- The company logo should be visible on every page.
- The header should include information about page number, total pages, chapter number and chapter name.

Fortunately, someone at our company has provided a company logo in Encapsulated PostScript format (`*.eps`), and a template header definition file. The template looks like this:

```
IFFIRST
18 28 Name of designer
ENDIF

IFNOTFIRST
183 20 <page>(<totalpages>)
90 24 Chapter <chapternumber> <chaptername>
ENDIF

EPSF 10 25 /home/lat/hello/tlog.eps
90 20 Name of SDL system
```

Action

1. The header symbol is not visible in the Organizer view. To make it visible, bring up the *View Options* dialog and select the *Header file* item and press *Apply* and *Close*.

2. To create a new and unique header for the SDL system, double-click on the header symbol in the Organizer view.

3. In the Edit dialog that follows, we should specify the template file under *Copy existing file*. In our case, someone has put the template file in `/home/lat/hello/headerTemplate.txt`.

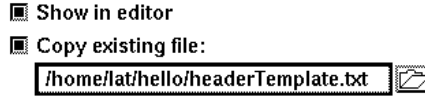


Figure 106: Specifying the template header file in the Add New dialog

4. When we close the Edit dialog, the text editor pops up with a copy of the template header file.

5. To get a unique header for our SDL system, we change the following things:

```
Name of designer -> Lars Tufvesson
Name of SDL system -> HelloWorld
```

6. When that is done, we save our specialized header file in the same directory as our SDL diagrams.

A Title Page

A title page is a plain text document that is placed first among all printed pages and documents.

Action

1. We should add a new plain text document to our SDL system. To make sure it is placed first, select the header symbol before invoking *Add New*.
2. In the Add New dialog, set the document type to *Plain Text* and set the document name to *TitlePage*.
3. When you close the dialog, the Text Editor pops up. Type in an appropriate title page text (like the one below) and save the text file in the same directory as the SDL diagrams.

```
******************
*System HelloWorld*
******************

A revolutionary SDL system software package
```

Chapters and Sub Chapters

We would like to divide our diagrams and documents in three chapters:

- The first chapter contains state matrices for the SDL process(es).
- The second chapter contains the SDL system.
- The third chapter is an appendix that contains the index of SDL entities.

The second chapter will be further divided into two sub chapters. Each chapter and sub chapter should have a chapter introduction text.

Action

1. We would like to place the first chapter directly after the title page text symbol. Select the title page text symbol and bring up the *Add New* dialog.
2. Set the document type to *Organizer > Chapter 1*, and the document name to *State Matrices*.
3. When you close the dialog, the Text Editor will pop up. Type in an appropriate chapter introduction text (like the one below) and save the text file.

```
This chapter contains two state matrices for the
SDL process: One with page numbers to the SDL
process and one with nextstate information.
```

We will produce the state matrices for this chapter in the next sub section. For now, we leave this chapter empty.

4. We introduce a second chapter for the SDL system. Select the *State Matrices* chapter symbol and bring up the *Add New* dialog. Set the document type to *Organizer > Chapter 1*, and the document name to *SDL system*. When you close the dialog, the Text Editor will pop up. Type in an appropriate chapter introduction text (like the one below) and save the text file.

   ```
   In this chapter, you will find the actual SDL
   system.
   ```

5. The SDL system chapter should have two sub chapters for the SDL process. To be able to begin a chapter between two pages in one SDL diagram, we must make page symbols visible in the Organizer. This is done in the *View Options* dialog by selecting *Page symbols* and pressing *Apply* and *Close*.

6. The first sub chapter should begin where the process diagram begins. Select the process diagram symbol and bring up the *Add New* dialog. Set the diagram type to *Organizer > Chapter 1.1*, and the document name to *First process page*. When you close the dialog, the Text Editor will pop up. Type in an appropriate chapter introduction text (like the one below) and save the text file.

   ```
   In this sub chapter, the first process page will be
   defined.
   ```

7. The second sub chapter should be placed between the two pages of the SDL process. Select the first process page symbol and bring up the *Add New* dialog. Set the diagram type to *Organizer > Chapter 1.1*, and the document name to *Second process page*. When you close the dialog, the Text Editor will pop up. Type in an appropriate chapter introduction text (like the one below) and save the text file.

   ```
   In this sub chapter, the second process page will be
   defined.
   ```

8. Finally, we should have an appendix chapter for the index of SDL entities. This chapter should be placed last in the Organizer view.

Select the last symbol in the Organizer and bring up the *Add New* dialog. Set the diagram type to *Organizer > Chapter 1*, and the document name to *Appendix*. When you close the dialog, the Text Editor pops up. Type in an appropriate chapter introduction text (like the one below) and save the text file.

```
The appendix contains an index of all SDL entities.
```

State Matrices for the SDL Process

The Organizer can with the help of the Analyzer generate a state overview information file (`*.ins`). The file is placed in the target directory. The Text Editor can display this file as state matrices.

Action

1. First, we should set the target directory to an appropriate directory where we can allow SDL Suite and TTCN Suite to create files. Bring up the *Set Directories* dialog and type in an appropriate target directory. When that is done, close the dialog.

2. Select the SDL system diagram and invoke the state overview information file generation with *Generate > State Overview*. The Analyzer will check your SDL system and produce some files in the target directory. One of these files is the `*.ins` file.

3. When the Analyzer is ready, the Text Editor will pop up, showing you the state matrices created from the information in the state overview file (`*.ins`). When you have examined the state matrices, exit the Text Editor.

4. The plain text symbol connected to the `*.ins` file should be placed in the first chapter instead of in the same chapter as the SDL system. To move the plain text symbol for the `*.ins` file to the correct location, select the symbol and use the up quick button as many times as required to get the symbol into the first chapter.

An Index of SDL Entities

We would like an index of SDL entities defined in our SDL system last in the printout. The Organizer can with the help of the Analyzer produce a cross reference file. The Index Viewer can present the information in the cross reference file as an index. To keep the size of the index small,

we filter out some SDL entities and we present the index in an as compact way as possible.

Action

1. Make sure that the target directory is set to an appropriate directory where the Analyzer can create some files.
2. Select the SDL system diagram symbol and press the *Generate Index* quick button. The Analyzer will check your SDL system and generate some files in the target directory. One of the files is named `*.xrf` and contains the cross reference information. The Index Viewer pops up when the Analyzer is ready, and presents the cross reference information as an index.
3. If you examine the index, you will notice that there are a lot of SDL entities defined in *package Predefined*. Let us assume that we are not interested in these SDL entities. To get rid of them, we bring up the *Filter Diagrams* dialog and select *package Predefined*. When the dialog is closed, our Index will be much smaller.
4. To get an even smaller index, bring up the *Index Options* dialog and set the index appearance to *Compact*.
5. Now, the index is ready for printing. Since there is no symbol in the Organizer view for the cross reference file, we have to leave the Index Viewer window open until we have completed the print operation.

Performing the Print Operation

Now, everything is in place to do the actual print operation.

Action

1. Make sure you have no selection in the Organizer. This can be done by clicking somewhere outside a symbol or text line. This is done to tell the Organizer that we want to print everything, and not just what we have selected.
2. Bring up the *Print* dialog.

3. Make sure that the following entries in the Contents area are selected:

   – Table of contents
   – Organizer View
   – SDL Interaction
   – Text Diagram
   – Index View

   No other entries in the Contents area should be selected.

4. Bring up the *Table of contents* setup dialog with the picture button close to the *Table of contents* entry. In the setup dialog, make sure that both *After initial text document* and *Only chapters in TOC* are both selected. Close the setup dialog.

5. Make sure that the header file check button is on.

6. Make sure that the settings in the Destination area are appropriate.

7. Click the *Print* button. We will get a printout, looking approximately as the figures below.

HelloWorld

Lars Tufvesson

```
********************
*System HelloWorld*
********************
A revolutionary SDL system software package
```

Figure 107: The title page

HelloWorld ii(12)
Chapter

Contents

Figure 108: The table of contents

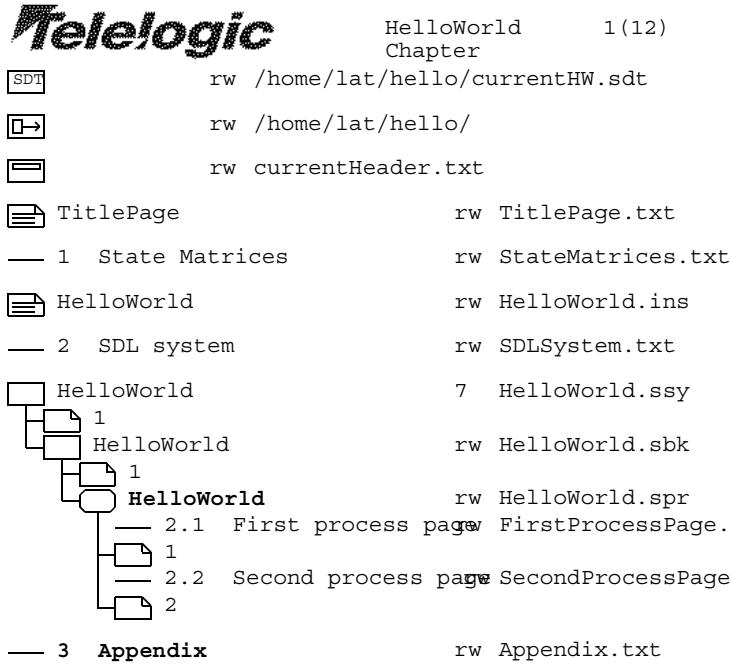
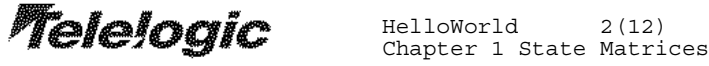


Figure 109: The Organizer view



This chapter contains two state matrices for the SDL process: One with page numbers to the SDL process and one with nextstate information.

Figure 110: The introduction to chapter 1

```
                                HelloWorld      3(12)
                                Chapter 1 State Matrices
Process HelloWorld: page numbers
States
a Start state
b waiting
c stop
          |  STATES
SIGNALS   |  a    b    c
          |-------------
-         |  8
Hello     |       8

Process HelloWorld: nextstates
States
a Start state
b waiting
c stop
          |  STATES
SIGNALS   |  a    b    c
          |-------------
-         |  b
Hello     |       c
```

Figure 111: The state matrices

```
                                HelloWorld      4(12)
                                Chapter 2 SDL System

In this chapter, you will find the actual SDL system.
```

Figure 112: The introduction to chapter 2

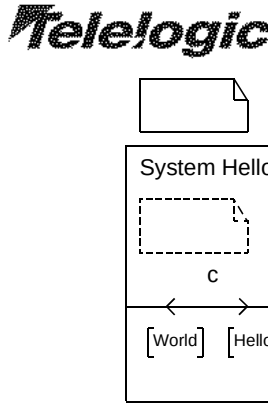


Figure 113: The SDL system diagram

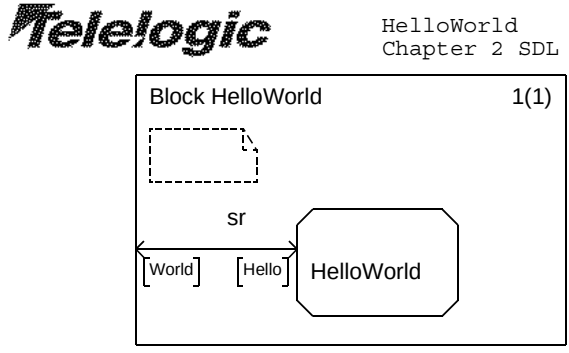


Figure 114: The SDL block diagram

This sub chapter contains the first part of the SDL process

Figure 115: The introduction to chapter 2.1

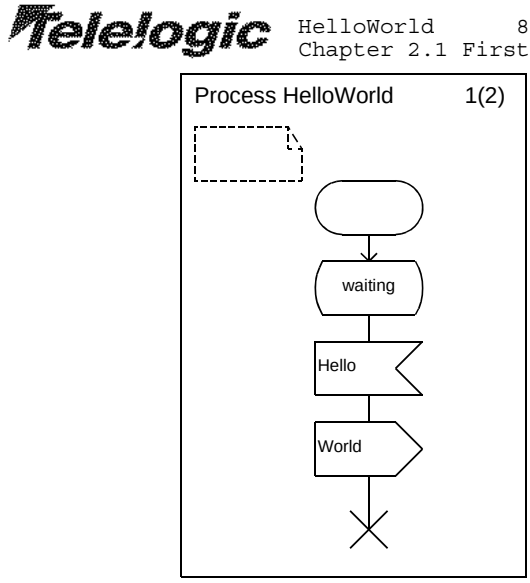


Figure 116: The first process page

This sub chapter contains the second part of the SDL process

Figure 117: The introduction to chapter 2.2

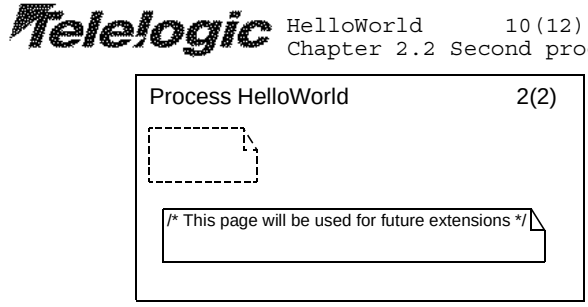


Figure 118: The second process page

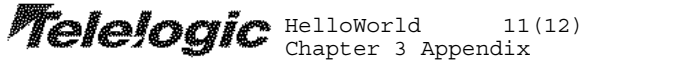


Figure 119: The introduction to the appendix

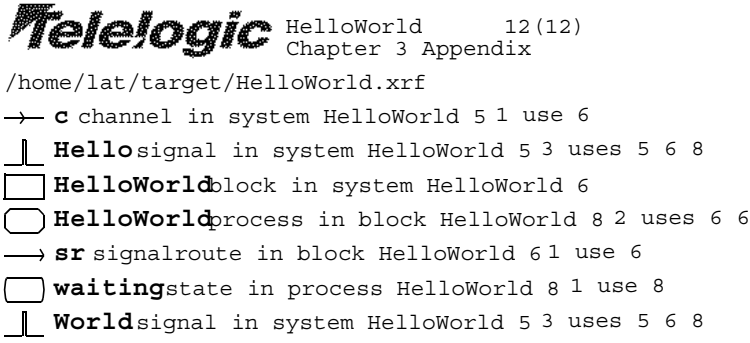


Figure 120: The index

Using Advanced Print Facilities

In the previous section, we took an ordinary SDL system and made enhancements to get a nice printout. In this section, we will look at how we can make the enhancements as permanent as possible. We would like to be able to load a system file, invoke the print operation and get a nice printout without changing any print settings.

Let us take a new look at all the changes we have made, to see how we can make a future print operation as painless as possible.

A Permanent Title Page

Every time the print dialog is invoked, the *Table of contents* setup option *After initial text document* will be turned on. This means that the title page will show up as a title page in future print operations also. No extra actions necessary here.

Permanent Chapters and Sub Chapters

The chapter symbols should stay where they were initially placed. Due to implementation reasons, chapter symbols placed between SDL page symbols might be a little more volatile than they should be. The position of a chapter symbol is determined by remembering the name of the next SDL page symbol. If you rename that SDL page symbol, the chapter looses its hook and may end up in the wrong position.

The chapter options that you can set in the *Chapter Options* dialog are reset every time SDL Suite and TTCN Suite are restarted. This means that if you want the start chapter number to permanently be anything other than 1.1.1.1, or you want to hide sub chapters in the table of contents, then you have to check the *Chapter Options* settings before each print operation.

Permanent State Matrices

State matrix options are not saved together with the state overview file (`*.ins`), but they are saved in the system file (*.sdt) and in print selection files (*.sel).

The state overview file (`*.ins`) is generated in the target directory. You may regard the target directory as a temporary directory where files can be erased after an SDL Suite or TTCN Suite session has ended. It might be cumbersome to regenerate a state overview file each time you want

to do a printout. In this situation, it might be a good idea to move the state overview file to the source directory, where your SDL diagram files resides. The file move is best done outside of SDL Suite, but in the Organizer view you have to reconnect the plain text symbol to the (moved) state overview file in the source directory.

A Permanent Index

Index options are not saved together with the cross reference file (*.xrf), but they are saved in the system file (*.sdt) and in print selection files (*.sel).

The cross reference file (`*.xrf`) is generated in the target directory. You may regard the target directory as a temporary directory where files can be erased after an SDL Suite or TTCN Suite session has ended. It might be cumbersome to regenerate a cross reference file each time you want to do a printout. In this situation, it might be a good idea to move the cross reference file to the source directory, where your SDL diagram files resides. The file move is best done outside of SDL Suite.

To get paper page references in the index, the index has to be printed from the Organizer together with the SDL system.

Performing a Permanent Print Operation

Every time the print dialog is invoked, the *Table of contents* setup option *Only chapters in TOC* will be set to off. This means that if you want only chapters in the table of contents, you have to turn this option on every time you want to print.

In a similar manner, every time SDL Suite and TTCN Suite is started, the *Table of contents* toggle button in the main print dialog is set to off. This means that every time you want to do a print, you have to check the *Table of contents* toggle button and turn it on, if it is off.

Print Selection Files and Print Selection Symbols

Most print, index and cross reference options are saved in the system file (*.sdt). This is often enough for most of your print needs. However, in some situations you might want to save more than one set of print settings: You want to easily switch...

- Header and footer information
- The set of documents/diagrams/pages that should be printed.

This can be done by saving a couple of print selection files with different print settings, and reference these files with print selection symbols in the Organizer.

1. To save print settings in a print selection file, bring up the Organizer print dialog, as you would for performing a normal print operation.
2. Adjust the settings in the dialog to what you want printed, both by changing what diagrams to print and by changing how the print will be performed (paper format etc.).
3. When all settings are correct, use the *Save* button to save the print selection file. In the dialog that appears, you can both decide the print selection file name to use, and if a print selection symbol should be created in the Organizer or not.

A print selection can be reused in several ways:

- Double-click the print selection symbol.
- Select the print selection symbol and invoke File > Print > Selected.
- Invoke File > Print > Selection File, and specify the print selection file in the dialog that appears.

In all these cases, the Organizer print dialog will appear, with settings according to the reused print selection file.

Chapter

6 *The Text Editor*

The Text Editor is the graphical tool that is used to create, edit, display and print text file documents.

The Text Editor can also be used as a state matrix viewer. One or several state-signal matrices can be presented for each process in an SDL system. The state matrices for an SDL system are presented as one read-only text file. As in any other SDL viewer tool, it is possible to navigate from an entity in a state matrix to the corresponding entity in the SDL system.

This chapter contains a reference manual to the Text Editor; the functionality it provides, its menus and windows.

Text Files

The Text Editor works with ASCII text files. The text editor can be used to view, edit and print text files. In addition to providing typical text editor functions, the Text Editor provides special support for embedding link endpoints into the text. For more information on links and endpoints, see chapter 9, *Implinks and Endpoints*.

> **Note:**
>
> The Text Editor is only available if the Organizer is started with the preference SDT*TextEditor set to “SDT”.

Endpoint Support

In the Text Editor, endpoints are user-defined, non-overlapping, contiguous ranges of characters, possibly spanning several lines of text, that are made visible to other tools, allowing you to define links between fragments in your text and endpoints in other documents.

Endpoints are displayed in the text editor by underlining the range of characters that are part of the endpoint.

Note that when you define an endpoint, you do not impose any particular restrictions on the text in the endpoint, meaning that you can still edit the text in the endpoint using the normal text editing functions to modify the endpoint’s contents. The text editor dynamically notifies other tools the changes you make to your endpoints as you make your changes, to make sure that the view of the endpoints in your file is always current.

Considerations when Using Text Endpoints

The requirement that endpoints must be non-overlapping means that you cannot create an endpoint inside another endpoint. Also, while an endpoint is allowed to span several lines, it is generally best to restrict endpoints to small, well-defined fragments of text on a single line, used as external reference points to the information in your files.

Also note that while the Text Editor visualizes endpoints by simply underlining the text in the endpoints, the endpoints are stored in your actual text files using special keywords before and after the characters in the endpoint. This means that it is not advisable to view or edit text documents with endpoints in a text editor that is not aware of the special significance of these keywords. If you do so, you should be careful so

as to not destroy information about the endpoints in the file, as this could cause links attached to these endpoints to break.

Another consequence of how the Text Editor stores the endpoint information is that if you are editing text documents that are intended as input to other tools (such as a C or C++ compiler) you should take care to place endpoints only within commented regions of text. This makes sure that the extra text inserted when storing the endpoints does not conflict with the allowed grammar of the type of file you are editing.

Examples

In the examples below you will find typical placements of text endpoints. These endpoints could then be the basis for creation of links into requirements, analysis, design, and specification documents, to name only a few possibilities.

Example 23 ----------

Improperly commented C header with endpoints:

```
/* Primary protocol packet */
struct Packet { ... };
```

Properly commented C header with endpoints:

```
/* Primary protocol packet */
struct Packet { ... };
```

Example 24 ----------

Improperly commented C++ header with endpoints:

```
class Base {

virtual char *name() const = 0;
        virtual int size() const = 0;
                          ...
}
```

Properly commented C++ header with endpoints:

```
// class Base
class Base {

virtual char *name() const = 0;  // get name
        virtual int size() const = 0;    // get size
                          ...
}
```

State Overview Files and State Matrices

The Organizer can produce a state overview file, see "State Overview" on page 130 in chapter 2, *The Organizer*. The Text Editor recognizes a state overview file by the extension `.ins`. If such a file is opened in the Text Editor, the Text Editor extracts information from the file and presents the information as a read-only text file containing state matrices. State overview files cannot be edited or saved in the Text Editor.

A state matrix is a textual table with the states in a process or procedure along one dimension and the signals in the same process or procedure along the other dimension. In the table, you will find the transitions for the process or procedure.

A state matrix may look like this: ("-" means no signal)

```
Process Main: nextstates

States
a Start state
b Game_Off
c Game_On
           |  STATES
SIGNALS    |  a    b    c
           |-------------
-          |  b
Newgame    |       c
Endgame    |            b
```

The Text Editor keeps track of how entities in the state matrix is related to the SDL system. This makes it possible to navigate back to the SDL system. To do so, place the text cursor close to an entity in a state matrix and bring up the pop-up menu for that entity and invoke one of the available menu choices.

A state overview file can be saved as a normal text file. This might be useful if you want to edit a state matrix. But by doing so, you will lose all the extra information associated with the state matrices that is not visible when viewing a state overview file in the text editor. You will no longer be able to:

- Navigate back to SDL.
- Get paper page references to SDL pages for transitions in the state matrix (read more about how this is done in "Advanced Print Facilities" on page 357 in chapter 5, *Printing Documents and Diagrams*).
- Decide the kind of information that should be presented for a transition in the state matrix.

Text Editor User Interface

The Text Editor window is depicted below. For a general description of the user interface, see chapter 1, *User Interface and Basic Operations*.

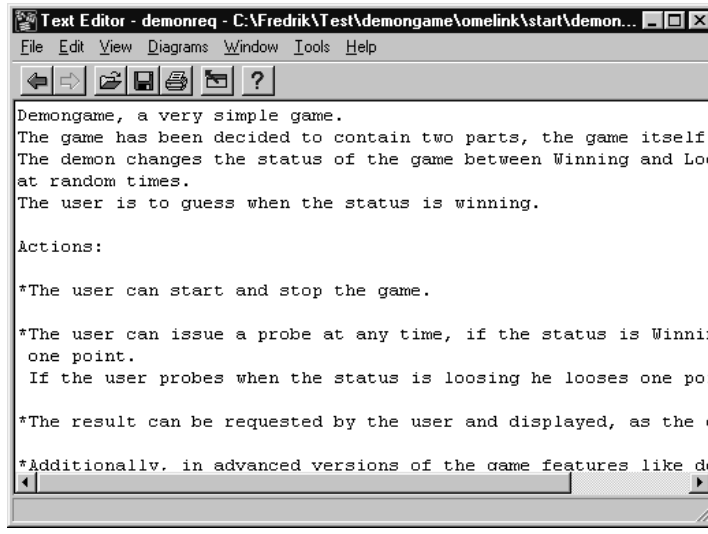


Figure 121: The Text Editor window

Text Area

Text in the Text Editor is displayed and printed using the non-proportional font face *Courier*, which is suitable for displaying ASCII texts such as source code.

While the screen font is fixed, you can change the font face used use for printing by editing the SDT preference SDT*PrintFontFamily.

Text Editing Functions

The following text editing functions are provided in the text window:

Text Selection

To select text you can either:

- Drag the pointer over the text.
- Place the cursor in the beginning or end of the text to be selected and then `<Shift>`-click where you want the selection to end.

Word Selection

Double-click a single word, delimited by spaces, to select it.

Line Selection

Triple-click a line of text to select it.

Text Editing

When you edit text, you can:

- Position the cursor by clicking or by using the arrow keys.
- Insert characters after the current position of the text pointer.
- Delete one character backward by pressing `<backspace>`.
- Delete one character forward by pressing `<delete>`.
- Replace selected text by typing.
- Delete selected text by pressing `<backspace>` or `<delete>`.

Undoing Text Modifications

The *Undo* command reverts the previous editing command. Several undo operations can be performed in sequence to undo sequences of editing commands.

Undo can negate the effects of all commands that alter the text in the Text Editor; however, the effect of user operations on endpoints and links cannot be undone by the undo command.

Depending on the nature of the last performed operation, the effects of the last text editing operation is undone, on a per editing operation. For undo purposes, the text editor recognizes the following operations:

- Typing a character
- Inserting a character or a fragment of text
- Deleting a character or a fragment of text

While having to undo each typed character individually may be somewhat inconvenient, a keyboard accelerator (`<Ctrl+Z>`, see "Keyboard Accelerators" on page 406) can speed up the process.

Note that since the text editor supports unlimited undo, it is not possible to use the undo operation to undo the last undo operation. Instead there is a keyboard accelerator (`<Ctrl+Y>`, see "Keyboard Accelerators" on page 406) which can be used repeatedly to undo the effects of a sequence of undo operations.

Editing Text Containing Endpoints

It is possible to edit text using normal operations even after endpoints have been embedded in your text. To unambiguously determine the effects of editing operations on endpoints, the Text Editor uses the character just before the position of your insertion cursor or your selection (the initial character) to determine if your editing affects an endpoint:

- If the initial character belongs to an endpoint (e.g. the insertion cursor is positioned just after an endpoint), inserted text will become part of the endpoint.
- If the initial character belongs to non-endpoint text (e.g. the insertion cursor is positioned just before an endpoint), the inserted text will not be included in an endpoint.

Note:

When deleting or overwriting text, you should be careful not to delete or overwrite all characters in an endpoint since this will remove the endpoint and the associated links.

Dealing with Consecutive Endpoints

When working with consecutive endpoints (i.e. two endpoints follow each other without any intervening non-endpoint text), it is impossible to insert extra characters between these endpoints using the normal text editing operations, since any text inserted between the two endpoints will be considered a part of the first endpoint.

If this particular case arises, you should use *Clear Endpoint* (see "Link > Clear Endpoint" on page 447 in chapter 9, *Implinks and Endpoints*) to remove some characters from one of the endpoints so that some non-endpoint characters become available between your endpoints.

Also note that while the text editor has no difficulty in dealing with consecutive endpoints, visual limitations will make it difficult to distinguish between the two endpoints. You should therefore avoid using consecutive endpoints if possible.

Menu Bar

The menu bar provides commands available in menus and menu choices for editing diagrams and pages of diagrams. Most the functionality that the Text Editor offers is contained within the commands from the menu bar. The Text Editor's menu bar provides the following menus:

- *File Menu*
- *Edit Menu*
- *View Menu*
- *Diagrams Menu*
- *Window Menu*
- *Tools Menu*
- *Help Menu*
  (see "Help Menu" on page 15 in chapter 1, *User Interface and Basic Operations*)

File Menu

The *File* menu contains the following menu choices:

- *New*
- *Open*
- *Save*
- *Save As*
- *Save a Copy As*
- *Save All*
- *Close Diagram*
- *Revert Diagram*
- *Print*
- *Exit*

The menu choices are described in "File Menu" on page 8 in chapter 1, *User Interface and Basic Operations*, except *Print* which is described in "The Print Dialogs in the SDL Suite and in the Organizer" on page 316 in chapter 5, *Printing Documents and Diagrams*.

Edit Menu

The *Edit* menu provides the following choices, which may or may not be dimmed depending upon whether or not some text is selected:

- *Undo*
- *Cut*
- *Copy*
- *Paste*
- *Paste As*
- *Clear*
- *Select All*

Undo

This command restores the last text editing operation. The following operations can be undone:

- *Typing, insertions and deletions.*
- *Cut*, *Paste*, *Paste As* and *Clear*

> **Note:**
>
> Undo does not restore endpoints or links, even though typing, insertions and deletions can destroy both endpoints and links

For more information on the operation of the *Undo* command, see "Undoing Text Modifications" on page 391.

Cut

Cut removes the selected portion of text from the text window and saves it in the clipboard buffer just as if a *Copy* has been made. *Cut* is only available if a portion of text has been selected. *Cut* is not available for state overview files (`*.ins`), because they are not editable.

Also see "Deleting an Object" on page 461.

Copy

Copy makes a copy of the selected portion of text in the clipboard buffer. The content of the text window is not affected. *Copy* is only valid if a portion of text has been selected.

When copying text to the clipboard buffer it becomes possible to paste into three different contexts:

- As simple text (without endpoint and link information) into applications that normally support cut/copy of text, such as terminal windows and other text editors.
- As text with endpoint and link information when pasting into one of the Text Editor windows, which means that endpoints and links are preserved.
- As a Paste As operation in all the Editors (including the Text Editor itself), provided that the copied text contained no endpoints or exactly matched an endpoint. See "Pasting a Text Fragment" on page 460 for more information.

Paste

Paste inserts the content of the clipboard buffer into the text window. The text in the clipboard buffer will be appended immediately following the cursor position, or replacing the selected text. *Paste* is only available if a portion of text has been cut or copied into clipboard buffer. *Paste* is not available for state overview files.

Also see "Pasting an Object" on page 461.

Paste As

Pastes the currently copied object (from the OM or Text Editor) as a text in the text window. The object is transformed and a link is optionally created between the copied object and the pasted text.

The Paste As dialog is opened. See "The Paste As Command" on page 448 for more information. *Paste As* is not available for state overview files (`*.ins`), because these files are not editable.

Clear

Clear removes the selected portion of text from the text window. The content of the clipboard buffer is not affected. *Clear* is only available if a portion of text has been selected. *Clear* is not available for state overview files (`*.ins`), because these files are not editable.

Also see "Deleting an Object" on page 461.

Select All

Select All selects all of the text contained in the text window.

View Menu

The *View* menu provides rescaling functions and access to various options that affect the behavior of the Text Editor. The following menu choices are available:

- *Window Options*
- *Editor Options*
- *State Matrix > Filter Processes*
- *State Matrix > Matrix Options*

Window Options

This menu choice issues a dialog where the presence of the tool and status bars can be set:

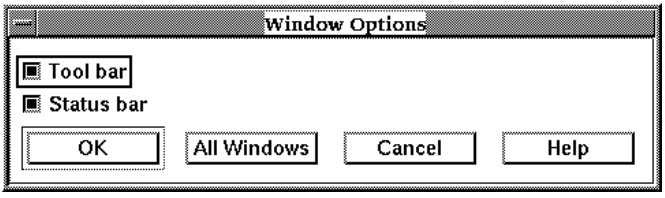


Figure 122: The Window Options dialog

- Clicking *OK* applies the options as defined in the dialog to the current window only.
- Clicking *All Windows* applies the options as defined in the dialog to all windows opened by the Text Editor.

Editor Options

This menu choice issues a dialog where the behavior of the Text Editor can be customized.

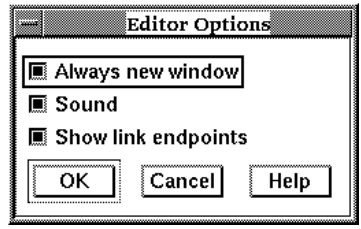


Figure 123: The Editor Options dialog

The options are controlled by toggle buttons. They are:

- *Always new Window*

  This option indicates whether or not a new window should be opened whenever the *New* or *Open* command or any command from the *File* menu is operated.

  The default behavior is not to open a new window.

- *Sound*

  This option indicates whether or not improper actions in the Text Editor, such as attempting to overlap symbols, should be brought to the user's attention by producing an alert sound.

  The default value for this option is on.

- *Show Link Endpoints*

  This option indicates whether link endpoints should be displayed by underlining the endpoint text.

Note:

Regardless of the setting of this option, link endpoints are never shown when printed.

The default value for this option is controlled by the TE*ShowLinks preference.

State Matrix > Filter Processes

This menu choice is only available for state overview files (`*.ins`).

This menu choice brings up a dialog that decides the group of processes that should be visible as state matrices. As default, all processes in the SDL system are visible. The dialog allows you to show selected processes, hide selected processes or show all processes.

State Matrix > Matrix Options

This menu choice is only available for state overview files (`*.ins`).

This menu choice brings up a dialog with options for state matrices. The available options are listed below.

Sort states

The states in the state matrices will be sorted in alphabetical order.

Sort signals

The signals in the state matrices will be sorted in alphabetical order.

Show page reference matrix

The page references refer to the printed page where the transition in question starts, i.e. where the SDL input symbol is placed. '-' means no signal.

Page references are normally not visible, they are replaced with '*'. The only chance you have to see page references instead of '*' is to print the SDL process together with the state overview file (`*.ins`) from the Organizer.

```
Page reference matrix for process Demon:

States
a Start state
b Generate

          |  STATES
SIGNALS   |  a    b
          |--------
-         |  *
T         |       *
```

Show nextstate matrix

For each transition, the possible nextstates are listed. ‘-’ means no signal.

```
Nextstate matrix for process Demon:

States
a Start state
b Generate

          |  STATES
SIGNALS   |  a    b
          |--------
-         |  b
T         |       b
```

Show signal sending matrix

For each transition, signals that might be sent from SDL output symbols during the transition are listed. ‘-’ means no signal or no output.

```
Signal sending matrix for Process Demon:

States
a Start state
b Generate

          |  STATES
SIGNALS   |  a    b
          |--------
-         |  -
T         |       1

1 = Bump
```

Show procedure call matrix

For each transition, procedures that might be called during the transition are listed. ‘-’ means no signal or no procedure call.

```
Process Demon: procedure calls

States
a Start state
b Generate

          |  STATES
SIGNALS   |  a    b
          |--------
-         |  -
T         |       -
```

Diagrams Menu

The *Diagrams* menu records all diagrams and pages that are opened by the Text Editor. The available menu choices are:

- *Back*
- *Forward*
- *<Diagram Name>*
- *List All*

Back

Select this menu choice to browse back to the document that was previously displayed in the window.

Forward

Select this menu choice to browse forward to the document that was displayed in the window before you selected *Back*.

<Diagram Name>

The last edited page always goes to the top of the list, and subsequently moves the other diagrams and pages down a position. A maximum of 9 open pages can be shown. A tenth one will be put at the top of the list, but any subsequent opening of a diagram or page will only show the last 9 that have been opened. Another option – *List All* (at the bottom of the list) – is available to list all the open diagrams in the Text Editor.

Each item in the menu provides information about the diagram name, possibly followed by the file it is stored on (the file information is missing if the diagram has never been saved). A diagram that is preceded by an asterisk (‘*’) denotes that it has been modified during the Text Editor session.

List All

This menu choice becomes available when a maximum of 9 open pages has been surpassed. When *List All* is selected, it provides a dialog containing all diagrams and pages that are currently open in the Text Editor:

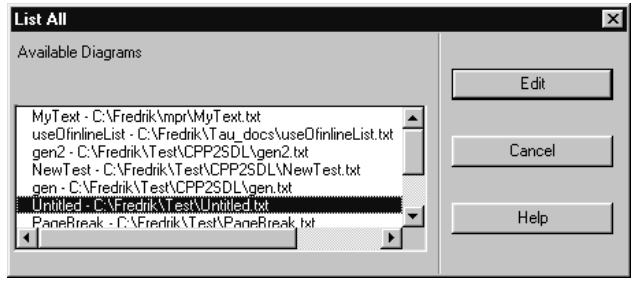


Figure 124: The List All dialog

Window Menu

The *Window* menu contains the following menu choices:

- *Close Window*
- *Entity Dictionary*

Close Window

This option closes the open window, **but**, not necessarily the diagram. If more than one editor window is opened, only the current window is closed and not the diagram. If the last open editor window is closed, the Text Editor will act as if *Exit* has been chosen, possibly in conjunction with a save of information (see "Close Diagram" on page 14 in chapter 1, *User Interface and Basic Operations*).

Entity Dictionary

Opens the Entity Dictionary window. See "The Entity Dictionary" on page 434 for more information.

When working with the Entity Dictionary in the Text Editor, note that the *Link > Create* command will not automatically create an endpoint on the current selection as is the case in the graphical Editors.

Tools Menu

The *Tools* menu contains the following menu choices:

- *Show Organizer*
- *Link > Create*
- *Link > Create Endpoint*
- *Link > Traverse*
- *Link > Link Manager*
- *Link > Clear*
- *Link > Clear Endpoint*

  (The *Link* commands are described in "Link Commands in the Tools Menus" on page 442 in chapter 9, *Implinks and Endpoints*.)

- *Go To Line*
- *Search*
- *Go To Source*
- *Show Transition*
- *Show Nextstate*
- *Show Output*
- *Show Call*
- *Show State*
- *Show Signal*

Go To Line

This menu choice moves the cursor to a line specified by the user. When selected, a dialog will appear querying the user where to move.

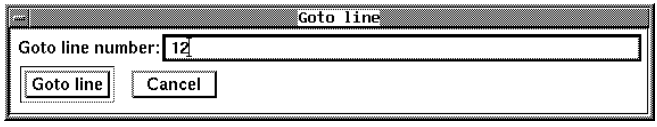


Figure 125: The Go To Line dialog

- *Goto line number*

  In this text field the user can specify a line number.

- *Goto line*

  Moves the cursor to the specified line and closes the dialog. This button will be dimmed if the *Goto line number* text field does not

contain a valid line number. If you have entered a line number that is larger than the number of lines in the document, the cursor is moved to the end of the document.

Search

This menu choice opens a dialog where you may search through the document for a text string and replace it with some other piece of text.

Figure 126: The Search dialog

Dialog Fields and Options

The *Search* dialog contains the following fields and options. Values from the previous invocation is used for settings when the dialog is used again.

- *Search for*

  The text to search for.

- *Replace with*

  Optionally, you can specify a text to replace the text searched for.

- *Search in*

  A read-only field showing the current status of the search. This field will display the text “Current document” until the search has ended, when it will show the text “End of document”.

- *Consider case*

  If this option is set, search is case sensitive.

- *Wildcard search*

  If this option is set, wildcard matching will be used when searching. In wildcard search, the asterisk character (’*’) matches a sequence

of zero or more characters of any kind. The backslash character ('\') escapes the character following it, which is useful if you want to search for asterisks.

- *Whole word search*

  If this option is set, search will only find whole words.

Dialog Buttons

When the dialog is first opened, all buttons except *Close* are disabled. Buttons will be enabled when appropriate, for instance the *Search/Restart Search* button will be enabled when a string is entered in the *Search for* text field.

When the dialog opens, searches always start at the top of the document.

- *Search/Restart Search*

  This button changed its name depending on the current search status. When it displays *Search*, it finds the next match in the text. When it displays *Restart Search* it will restart the search from the top of the document when pressed.

- *Replace & Search*

  Replaces the current match with the replace string and searches for the next match.

- *Replace All*

  Replaces all subsequent occurrences of the search string with the replace string.

Go To Source

This command takes the currently selected text fragment and attempts to interpret it as an SDT Reference and display it in the appropriate editor. The selection must include the selected reference exactly. This command corresponds to *Go To Source* in the Organizer.

The syntax of the graphical references used in the Text Editor is described in chapter 18, *SDT References*.

Show Transition

This operation is only available for state overview files (`*.ins`), and only when the text cursor is placed close to a transition in a state matrix.

An SDL Editor window is popped up with the SDL input symbol corresponding to the start of the transition currently in focus in the Text Editor.

Show Nextstate

This operation is only available for state overview files (`*.ins`), and only when the text cursor is placed close to a transition in a state matrix.

Show Nextstate is a sub menu. The menu choices in the sub menu have the names of the states found in all nextstate symbols that ends the transition currently in focus in the Text Editor. When a nextstate name is selected, an SDL Editor window pops up, showing the nextstate symbol.

Show Output

This operation is only available for state overview files (`*.ins`), and only when the text cursor is placed close to a transition in a state matrix.

Show Output is a sub menu. The menu choices in the sub menu have the names of all signals that might be sent from an SDL output symbol during the transition currently in focus in the Text Editor. When a signal is selected, an SDL Editor window pops up, showing the output symbol with the signal sending.

Show Call

This operation is only available for state overview files (`*.ins`), and only when the text cursor is placed close to a transition in a state matrix.

Show Call is a sub menu. The menu choices in the sub menu have the names of all procedures that might be called during the transition currently in focus in the Text Editor. When a procedure call is selected, an SDL Editor window pops up, showing the procedure call symbol.

Show State

This operation is only available for state overview files (`*.ins`), and only when the text cursor is placed close to a state or transition in a state matrix.

Show State is a sub menu. The menu choices in the sub menu represents all the places where SDL state symbols can be found for the state currently in focus in the Text Editor. If a transition is currently in focus in the Text Editor, the *Show State* operation operates on the start state for the transition. When a menu choice is selected, an SDL Editor window pops up, showing the corresponding state symbol.

Show Signal

This operation is only available for state overview files (`*.ins`), and only when the text cursor is placed close to a signal or transition in a state matrix.

When this operation is invoked, an SDL Editor window is popped up with the signal declaration corresponding to the signal in focus in the Text Editor. If a transition is in focus in the Text Editor, the operation operates on the signal that triggers the transition.

Pop-Up Menu

Open the pop-up menu by right-clicking or pressing `<F2>`. The pop-up menu remains opened until you click the mouse or press `<Esc>`.

Pop-Up Menu in Text Documents

In text documents, the pop-up menu contains some of the menu choices in the *Edit* and *Tools* menus.

In state overview files (`*.ins`), the pop-up menu contains the state overview menu choices in the *Tools* menu. The menu choices are different depending on if a state, signal or transition is closest to the cursor position.

Keyboard Accelerators

In addition to the standard keyboard accelerators, described in "Keyboard Accelerators" on page 35 in chapter 1, *User Interface and Basic Operations*, the Text Editor includes the following:

Accelerator	Command / functionality
`Ctrl+L`	Redraw window.

Accelerator	Command / functionality
`Ctrl+Y`	Redo (i.e. undo last undo)
`Ctrl+1`	Show Organizer
`<Delete>`	Clear

Chapter 6 The Text Editor

Chapter

7

Emacs Integration

(UNIX Only)

On UNIX, Emacs can be used as an external text editor. By configuring SDL Suite and TTCN Suite to use Emacs for handling text documents, the Organizer can interact with Emacs in the same way as with the other Editors.

Handling of SOMT implementation links is supported.

This chapter contains a user manual to Emacs in SDL Suite and TTCN Suite; the functionality it adds and the new commands.

Overview

Introduction

The implementation works with GNU Emacs version 19.31 or later, and only **on UNIX** systems. It does **not** work with any Emacs for Windows.

It is assumed that you are familiar with Emacs terminology and how to use Emacs. Some experience with installing Emacs Lisp packages is also required when setting up the integrated environment.

Text Document Handling

By setting preferences to use Emacs as text editor in SDL Suite and TTCN Suite, text editing support in Emacs is offered both from the Organizer and editors. Text document handling is implemented in the Lisp package “sdtemacs”.

In the Organizer, text documents are identified by the document type “Text Plain” when you add new or existing documents. A new Emacs instance will always be started, even though there may exist one already (started from outside SDL Suite and TTCN Suite).

Link Handling

The Lisp package “sdtlinks” defines the minor mode *SDTlinks* which supports SOMT implementation links in text documents. This gives the possibility to handle link endpoints and follow the links defined for these endpoints. The support for link handling is optional.

When the *SDTlinks* mode is active, endpoints are displayed with a typeface differing from the plain text. Type faces for endpoints with and without links are configurable, see “Type Faces for Endpoints” on page 413.

To be able to represent endpoints in text files, the file contents differ from what is seen in the Emacs buffer containing a file with endpoints. The file format for endpoints look like this:

```
<SDT_LINK_ENDPOINT_BEGIN anchor >text<SDT_LINK_ENDPOINT_END>
```

where `anchor` is an integer uniquely identifying the endpoint and `text` is the endpoint text. If the *SDTlinks* mode is inactive, this raw file contents is displayed in the Emacs buffer.

Caution!

When the SDTlinks mode is *inactive*:

Do not alter any text representing endpoints (described in "Link Handling" on page 410).

Do not duplicate text containing endpoints since this will yield unpredictable results as each endpoint is uniquely identified in a file.

As a general principle, Undo is not available for the link handling commands.w

Installation

Text Document Handling

To be able to use Emacs as an external editor in SDL Suite and TTCN Suite:

1. Put the file `sdtemacs.el` in a Lisp directory known to Emacs (`load-path`). By default, the file is installed in `$telelogic/include/lisp`.
2. Byte-compile `sdtemacs.el` for increased execution speed.
3. Put the line:
   `(require 'sdtemacs)`
   in your `~/.emacs` file or in the file `default.el` in the `../lisp` directory of the Emacs distribution.

Make sure that the environment variable `$telelogicbin` is correctly set.

Link Handling

To be able to use link handling functions in Emacs:

1. Put the file `sdtlinks.el` in a Lisp directory known to Emacs (`load-path`). By default, the file is installed in `$telelogic/include/lisp`.
2. Byte-compile `sdtlinks.el` for increased execution speed.

3. Put the line:
   ```
   (require 'sdtlinks)
   ```
   in your `~/.emacs` file or in the file `default.el` in the `../lisp` directory of the Emacs distribution.

Sdtlinks requires sdtemacs, that is, you must install sdtemacs in order to use sdtlinks.

Installation Example

An example of using both packages:

```
(setq load-path (cons (substitute-in-file-name
"$telelogic/include/lisp") load-path))
(require 'sdtemacs)
(require 'sdtlinks)
```

Preferences

In the Preference Manager, the following parameters should be set:

Parameter	Value
SDT*TextEditor	Emacs
SDT*EmacsCommand	The command that starts GNU Emacs from a UNIX shell (the default is “emacs”).

Customizing

Type Faces for Endpoints

The type face for endpoints without associated links is called `sdtlinks-endpoint-face`. The default appearance is underlined, blue text.

The type face for endpoints with associated links is called `sdtlinks-endpoint-with-links-face`. The default appearance is underlined, bold, blue text.

These type faces can be redefined.

Example 25 ----------

An example to make endpoints without associated links appear as bold, red text:

```
...
(make-face 'sdtlinks-endpoint-face)
(make-face-bold 'sdtlinks-endpoint-face nil t)
(set-face-foreground 'sdtlinks-endpoint-face "red")
...
(require 'sdtlinks)
```

Maximum Number of Endpoints in a Document

Due to the nature of the execution environment for Emacs, only a finite number of endpoints are manageable in a document. By default, this limit is set to 1000 endpoints, which should be sufficient for most situations. If, however, a larger number is required, this can be achieved by setting the variable `sdtemacs-max-no-of-endpoints`.

Example 26 ----------

An example to allow 1500 endpoints:

```
...
(setq sdtemacs-max-no-of-endpoints 1500)
...
(require 'sdtlinks)
```

Emacs Commands

This section lists and describes the Emacs commands and key bindings associated with the integration with SDL Suite and TTCN Suite. Both new commands and the affected ordinary (existing) commands are described.

Text Document Handling

New Commands

Connect

`M-x sdtemacs-connect` connects to SDL Suite and TTCN Suite. Automatic connection can be performed when starting Emacs by providing the UNIX command switch `post` (`emacs -post`). This is done when SDL Suite and TTCN Suite starts Emacs.

Disconnect

`M-x sdtemacs-disconnect` disconnects from SDL Suite and TTCN Suite. All buffers taking part in the interaction will be unloaded after asking about saving modified buffers. Exiting Emacs will disconnect from SDL Suite and TTCN Suite.

Show Organizer

`M-x sdtemacs-show-organizer` opens or raises the Organizer Main window.

Show SDT Reference

`M-x sdtemacs-show-reference` shows an SDT reference in an appropriate editor. This requires that the reference text is selected or that the point is placed inside it.

Affected Ordinary Commands

Exit Emacs (C-x C-c)

Disconnects from SDL Suite and TTCN Suite before exiting.

New File (C-x b <new buffer>, C-x C-f <new file>)

The new buffer is notified to SDL Suite and TTCN Suite and takes part in the interaction with SDL Suite and TTCN Suite.

Open File (C-x C-f <filename>)

The loaded buffer is notified to SDL Suite and TTCN Suite and takes part in the interaction with SDL Suite and TTCN Suite.

Save (C-x C-s) and Save As (C-x C-w <filename>)

The buffer is notified as saved to SDL Suite and TTCN Suite.

Close File (C-x k)

The buffer is notified as unloaded to SDL Suite and TTCN Suite.

First Modification of Buffer Text

The buffer is notified as modified to SDL Suite and TTCN Suite.

Link Handling

The minor mode *SDTlinks*, which implements the link handling commands, requires that Emacs is connected to SDL Suite and TTCN Suite, and will try to connect when switched on if this is not the case. All buffers loaded when connected will automatically have this mode active.

New Commands

SDT Links Mode

`M-x sdtlinks-mode` toggles the minor mode SDTlinks. This is useful if performing editing operations that cannot be done without unintentionally deleting endpoints, e.g. moving (*Cut* and *Paste*) a region of text containing endpoints.

Caution!

When SDTlinks is *inactive*:

Do not alter any text representing endpoints (described in "Link Handling" on page 410).

Do not duplicate text containing endpoints since this will yield unpredictable results as each endpoint is uniquely identified in a file.

Create Endpoint

`M-x sdtlinks-create-endpoint (C-c C-c)` creates a new endpoint that will be marked with a special type face in the text (see "Type Faces for Endpoints" on page 413) and notified to SDL Suite and TTCN Suite. This command requires that the region of text to convert to an endpoint is selected, i.e. both mark and point must be set. Endpoints are not allowed to overlap and must consist of at least one character.

Delete Endpoint

`M-x sdtlinks-delete-endpoint (C-c C-d)` deletes an endpoint. The endpoint text will be converted to plain text and the deletion will be notified to SDL Suite and TTCN Suite. All links associated with the endpoint will be cleared. This command requires that point is placed inside the endpoint text.

Follow Link

`M-x sdtlinks-follow-link (C-c C-f)` follows one of the links associated with the selected endpoint. This results in displaying the endpoint on the other end of the link in an appropriate editor. If only one link is associated with the endpoint, this link is followed. Otherwise, a choice between the associated links is offered in a popup menu:

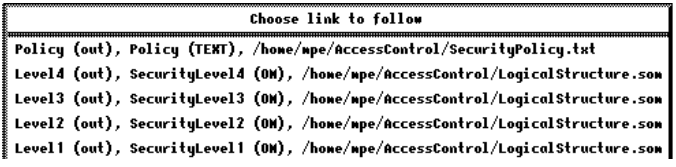


Figure 127 Choose link to follow

Each menu item, representing a link, is of the format:

```
<Link name>(<Direction>), <Other endpoint>(<Type>), <File>
```

where:

```
<Link name>      = The name of the link
<Direction>      = The direction of the link: 'in' or 'out'
<Other endpoint>= The name of the other endpoint of the link
<Type>           = The type of the other endpoint of the link
<File>           = The name of the file (including the path)
                   in which the other endpoint resides
```

This command requires that point is placed inside the text of an endpoint marked as having associated links, see "Type Faces for Endpoints" on page 413.

Show Endpoint in Link Manager

`M-x sdtlinks-show-endpoint-in-link-manager (C-c C-s)` opens the Link Manager and shows the selected endpoint. This requires that point is placed inside the endpoint text.

Affected Ordinary Commands

Modification of Endpoint Text

The endpoint is notified as modified to SDL Suite and TTCN Suite. When point is placed directly after the endpoint text, added characters will become part of the endpoint text.

Cut/Clear (C-w, C-k, M-k and many more)

For all endpoints within the region, *Delete Endpoint* (see "Delete Endpoint" on page 416) is executed.

Copy Region (M-w)

Endpoints in the copy of the region is represented as plain text.

Insert File (C-x i <filename>)

For all endpoints in the inserted file, *Create Endpoint* (see "Create Endpoint" on page 416) is executed. Associated links are not handled.

If point is within an endpoint and the file to insert have endpoints, the operation will fail since overlapping endpoints are not allowed.

Undo (C-x u, C-_)

No undo information is recorded for the link handling commands. Consequently, these commands cannot be undone.

Command Summary

Command	Description
`M-x sdtemacs-connect`	“Connect” on page 414
`M-x sdtemacs-disconnect`	“Disconnect” on page 414
`M-x sdtemacs-show-organizer`	“Show Organizer” on page 414
`M-x sdtemacs-show-reference`	“Show SDT Reference” on page 414
`M-x sdtlinks-mode`	“SDT Links Mode” on page 415
`M-x sdtlinks-create-endpoint (C-c C-c)`	“Create Endpoint” on page 416
`M-x sdtlinks-delete-endpoint (C-c C-d)`	“Delete Endpoint” on page 416
`M-x sdtlinks-follow-link (C-c C-f)`	“Follow Link” on page 416
`M-x sdtlinks-show-endpoint-in-link-manager (C-c C-s)`	“Show Endpoint in Link Manager” on page 417

Chapter

8 *MS Word Integration*

(Windows only)

In Windows, Microsoft Word is integrated with SDL Suite and TTCN Suite and MS Word documents are handled by the Organizer.

Handling of SOMT implementation links is supported.

This chapter contains a user manual to MS Word in SDL Suite and TTCN Suite; the functionality it adds and the new menu.

Overview

Introduction

The implementation works with MS Word 2000 (MS Office 2000) and MS Word 2003 (MS Office 2003) **in Windows**. It is assumed that you are familiar with MS Word terminology and how to use MS Word. Some experience with MS Word templates and macros is also required when setting up the integrated environment.

MS Word Document Handling

Support for MS Word document editing is available in the Organizer. When connecting MS Word to the Organizer, the process `SdtWord` will be started in the background.

In the Organizer, MS Word documents are identified by the document type “Text Word” when adding new or existing documents.

In the Organizer, MS Word document are represented with the icon shown to the left. Unlike other document types, when a MS Word documents becomes modified, its icon is not marked as “dirty”.

A new MS Word documents is always named “Document<x>”, where <x> is an integer, regardless of what name it was given in the Organizer.

Link Handling

SOMT implementation links in MS Word documents are supported. This gives the possibility to handle link endpoints and follow the links defined for these endpoints.

An MS Word endpoint icon in the Organizer looks the same as a document icon.

In MS Word, endpoints are marked with special character formats. The appearance for endpoints without associated links is underlined, blue text. The appearance for endpoints with associated links is double underlined, bold, blue text.

Endpoints in MS Word documents are represented with bookmarks whose names are of the format “SDT_<x>”, where <x> is an integer uniquely identifying the endpoint.

Undo is not available for the link handling commands.

Installation

MS Word

To be able to use MS Word with SDL Suite and TTCN Suite, do as follows:

For MS Word 7.0

1. Put the template file `sdt95.dot` (available in `<Installation Directory>\bin\wini386\sdt95.dot`) into the folder `\MSOffice\Winword\Startup`. This means that the template becomes global and the SDL Suite and TTCN Suite commands will be available in MS Word all the time.

2. Some macros are template dependent, which requires them to be defined for the template on which the document is based. The macros to be copied from `sdt95.dot` to your document templates are:

   ```
   AutoClose
   AutoNew
   AutoOpen
   ```

   Consult the MS Word documentation on how to copy the macros to your templates.

   The macros are (re)definitions of possibly existing MS Word macros. If you already have them defined in your document templates, you must add the code to your own macros.

For MS Word 8.0

There is an automatic installation program available in `<Installation Directory>\bin\wini386\sdtwordinstall.exe`. You can also make a manual installation:

1. Put the template file `sdt97.dot` (available in `<Installation Directory>\bin\wini386\sdt97.dot`) into a temporary folder.

2. Change the file protection from read-only to read/write.

3. Open this `sdt97.dot` in MS Word and copy the modules:

   ```
   SDTAutomacros
   SDTMisc
   ```

   to the template `normal.dot`. Make sure there are no conflicts with other automacros in `normal.dot`.

4. Delete the copied modules from `sdt97.dot`.

5. Close MS Word and move the `sdt97.dot` copy into the folder `\Microsoft Office\Office\Startup`.

6. Add a reference (in the MS Word VB-editor) from `normal.dot` to `sdt97.dot`.

The Startup Folder

You should check that MS Word points at your start-up folder by choosing *Tools* > *Options*. In the dialog that appears, select *File Locations* and check where the file type *Startup* is pointing. It should point at the directory where you placed `sdt95.dot` or `sdt97.dot`.

> **Note:**
>
> Do not use `sdt95.dot` or `sdt97.dot` as a document template. It only contains MS Word macros and should be used solely as a global template.

Preferences

In the Preference Manager, the following parameter should be set:

Parameter	Value
SDT*WordCommand	The command that starts MS Word (the default is “Winword.exe”).

Make sure that the path points to its directory.

MS Word Menu Commands

This section lists and describes the MS Word menu commands associated with the integration with SDL Suite and TTCN Suite. Both new commands and the affected ordinary (existing) commands are described.

New Menu Commands

The new commands are accessible from a menu in MS Word called *SDT*. It contains the following menu choices:

- *Connect to SDT*
- *Disconnect from SDT*
- *Show Organizer*
- *Show Reference*
- *Create Endpoint*
- *Delete Endpoint*
- *Follow Link*
- *Show Endpoint in Link Manager*
- *Update Endpoints*

Connect to SDT

Connects to SDL Suite and TTCN Suite. This is done automatically when starting MS Word.

Disconnect from SDT

Disconnects from SDL Suite and TTCN Suite. This is done automatically when exiting MS Word.

Show Organizer

Opens or raises the Organizer.

Show Reference

Shows the selected SDT reference in an appropriate editor. This requires that the user has selected the reference text or placed the cursor inside it.

Create Endpoint

Creates a new endpoint that will be marked with a special character format in the text (see "Link Handling" on page 420) and notified to SDL Suite and TTCN Suite. This command requires that the user has selected the region of text to convert to an endpoint. Endpoints are not allowed to overlap and must consist of at least one character.

Delete Endpoint

Deletes an endpoint. The endpoint text will be converted to plain text and the deletion will be notified to SDL Suite and TTCN Suite. All links associated with the endpoint will be cleared. This command requires that the user has selected or placed the cursor inside the endpoint text.

Follow Link

Follows one of the links associated with an endpoint. This results in displaying the endpoint on the other end of the link in an appropriate editor. If only one link is associated with the endpoint, this link is followed. Otherwise the user is offered to choose between the associated links in a dialog.

This command requires that the user has selected or placed the cursor inside the text of an endpoint marked as having associated links.

Show Endpoint in Link Manager

Opens the Link Manager and shows the selected endpoint. This requires that the user has selected or placed the cursor inside the endpoint text.

Update Endpoints

When applying styles and character formats to text containing endpoints, the endpoint character format may be changed and thus it becomes impossible to tell where the endpoints are located. By using this command all endpoints in the active document will have the endpoint character format applied to them.

Furthermore, when deleting text containing endpoint using the keyboard (<Delete>) instead of menu choices, or pasting text with endpoints from other documents, this is not notified to SDL Suite and TTCN Suite. By executing this command, a synchronization with the Link Manager takes place in order for it to have a consistent view of the endpoints in the active document.

Affected File Commands

Exit

Disconnects from SDL Suite and TTCN Suite before exiting.

New

The new buffer is notified to SDL Suite and TTCN Suite and takes part in the interaction.

Open

The loaded buffer is notified to SDL Suite and TTCN Suite and takes part in the interaction.

Save and *Save As*

The buffer is notified as saved to SDL Suite and TTCN Suite.

Close

The buffer is notified as unloaded to SDL Suite and TTCN Suite.

Affected Editing Commands

Modification of Endpoint Text

The endpoint is notified as modified to SDL Suite and TTCN Suite.

> **Note:**
> In MS Word, inserted text take the same character format as the character that precedes it. Added characters in front of the endpoint text will not take the character format of the endpoint text until the endpoint is updated (see "Update Endpoints" on page 424).
> MS Word 7.0 only: When the cursor is placed directly after the endpoint text, added characters will **not** become part of the endpoint text, even though the character format will indicate so.

Clear

All endpoints within the selection will be deleted. See "Delete Endpoint" on page 424.

Cut

All endpoints within the selection will be deleted. See "Delete Endpoint" on page 424. Endpoints in the copy of the selection is represented by plain text.

Copy

Endpoints in the copy of the selection is represented by plain text.

Paste

Endpoints (bookmarks) not already defined in the document, in which the Paste operation is done, will be created. However, the text region is still plain text and the endpoints have to be updated in order to apply the endpoint character format to the new endpoints.

Undo

No undo information is recorded for the link handling commands. Consequently, these commands cannot be undone.

Chapter

9 *Implinks and Endpoints*

Implementation links and endpoints are important concepts in the SOMT method (described in the SOMT Methodology Guidelines starting in chapter 69 in the User's Manual). This chapter describes the tool support for implinks and endpoints. The following topics are covered:

- **Link concepts**
- **How endpoints and links are visualized**
- **How to create links**
- **The Entity Dictionary window**
- **Link commands in menus**
- **The Paste As command**
- **The Link Manager tool**

Link Concepts and Overview

The SDL Suite supports creating and maintaining links between different objects in a system. Such links are used to show relations between objects in different documents. Objects that may be linked are:

- Text fragments in text documents
- Graphical objects and symbols in OM, SC, HMSC, SDL and MSC diagrams (but **not** text fragments in such diagrams)
- Documents in the Organizer structure.

Implinks

The link concept in the SDL Suite is designed to support the SOMT method, further described in the SOMT Methodology Guidelines starting in chapter 69 in the User's Manual. In SOMT, an important relation is that one object can be seen as an implementation of another object. For this reason, links between objects are often referred to as implementation links, or *implinks*. For instance, a textual object in the requirement analysis may be implemented as an object class in the system analysis, and later as a process type in the design model.

Implinks are the result of a design decision taken during the development of a system. Using implinks enables traceability between the different models and phases, so that the usage of a particular object or concept can be followed from requirements all the way down to code. Another important aspect of implinks is that they facilitate consistency checks between the different models.

Links and Endpoints

A link has two *endpoints*, one at each of the objects that are linked together. Endpoints can be created for objects without creating a link, i.e. endpoints are entities separate from links. When a link is created, endpoints are created automatically, if they do not already exist. It is possible to have any number of links connected to an endpoint. Endpoints and links can be created from the Organizer and the editors.

Links are bidirectional, i.e. they can be traversed (followed) in both directions. Even so, a link has a "default" direction, defined when the link is created, to indicate the intended direction. This means that an endpoint is either a logical "from" endpoint, or a logical "to" endpoint.

A link has a name and optionally a comment. The name indicates the type of link, e.g. “implementation link”, and the comment is used to describe the link.

Link File

Information about endpoints and links is stored in a central *link file* (extension `.sli`), which is referred to from the Organizer’s system file. This approach makes it easy to get an overview of existing endpoints and links, and to make consistency checks. The link file is saved whenever the system is saved. The link file, and its defined links, are managed by a dedicated tool, the Link Manager (see “Tool Support and Operations” on page 432).

Information about endpoints are also stored in the individual documents. However, link information is only stored in the link file.

Local Link File

To make it easier to use the endpoint and link features in a multiuser environment, the concept of a *local link file* is provided. This is a personal link file for one user, storing all changes made to endpoint and link information compared to the global file, the *master link file*. A controlled merge operation is provided to update the master link file with the local link file information. These operations are available as services in the SDL Suite and TTCN Suite Public Interface (see *“Link File Services” on page 552 in chapter 11, The Public Interface*).

Visualization of Endpoints and Links

By default, endpoints are marked in a special way in the editors and in the Organizer. The markers are slightly different depending on whether or not any links are connected to the endpoint.

Endpoints in Graphical Editors

In the SDL Editor and the SDL Suite diagram editors, the endpoint marker is a small triangle in the upper left corner of the object's enclosing rectangle. The triangle is filled if the endpoint has any links connected to it.

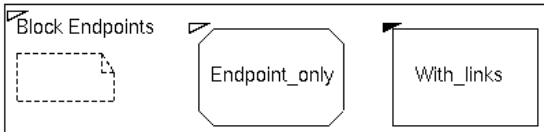


Figure 128: Endpoints with and without links

For lines in the SDL, OM. SC and HMSC diagrams, the marker appears on the name or signal list associated with the line. In SDL diagrams, it is also possible to create endpoints on other text elements associated with the line.

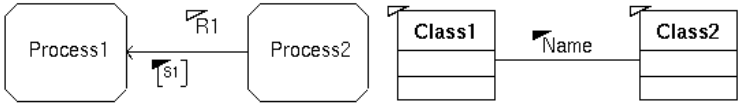


Figure 129: Endpoints on lines and text attributes

For lines in MSC diagrams (messages, timers, create requests), the marker normally appears at the "start" end of the line.

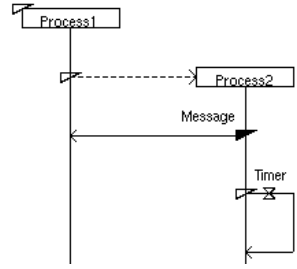


Figure 130: Endpoints on MSC lines

You can hide the endpoint markers by changing the option *Show Link Endpoints* in the *Editor Options* in the *View* menu, or by setting the editor preference ShowLinks to off.

Note:

Endpoint markers are never shown when printing a diagram.

Endpoints in Text Editors

In the Text Editor, endpoints are shown as underlined text, regardless if they have links connected to them or not. The endpoint text can be shown as normal text by changing the option *Show Link Endpoints* in the *Editor Options* in the *View* menu, or by setting the Text Editor preference ShowLinks to off.

In the Emacs editor **(on UNIX)**, endpoints without links are by default shown as blue, underlined text, whereas endpoints with links are shown as bold, blue, underlined text. The default font faces can be changed; see "Type Faces for Endpoints" on page 413 in chapter 7, *Emacs Integration*.

In MS Word **(in Windows)**, endpoints without links are shown as blue, underlined text, whereas endpoints with links are shown as bold, blue, double underlined text.

Endpoint text is always shown as normal text when printing a text document.

Endpoints in the Organizer

In the Organizer, the endpoint marker appears in the same way as in the graphical editors, i.e. a triangle in the upper left corner of the document icon.

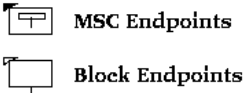


Figure 131: Endpoints in the Organizer.

The endpoint markers are always shown in the Organizer, and they are also shown when printing the Organizer window.

Tool Support and Operations

Operations on endpoints and links are supported in the Organizer, the diagram editors, the Text Editor and the Link Manager.

In the editors, the possible operations are identical and can be found in the *Link* submenu of the *Tools* menu (see "Link Commands in the Tools Menus" on page 442). The operations include:

- Creating and deleting an endpoint
- Creating and deleting a link
- Traversing a link (bringing the other endpoint into view)
- Opening the Link Manager

In the Organizer, the available operations in the *Tools* menu are limited to creating and deleting an endpoint, and opening the Link Manager.

In the editors, it is also possible to open the Entity Dictionary from the *Window* menu. The Entity Dictionary lists all endpoints in the system, together with the documents making up the Organizer structure. It is mainly intended for re-use of entity names, but also supports creating links. See "The Entity Dictionary" on page 434.

The Link Manager shows all endpoints and links in the system. It supports the following main operations:

- File operations on the link file
- Creating and deleting links
- Editing a link's direction, name and comment
- Performing consistency checks on endpoints and links

It is not possible to create and delete endpoints in the Link Manager. For more information about the Link Manager, see "The Link Manager" on page 462.

Creating Links

There are basically three different ways to create links:

1. Manually, by linking together two endpoints.

   This operation requires two already existing endpoints. The endpoints may have been created manually, or as an effect of creating a link earlier.

   This operation is only available through the use of the Link Manager. See "Create Link" on page 468.

2. Manually, by linking together an endpoint and a selected object.

   This operation requires an already existing endpoint (selected in the Entity Dictionary), and a selected object in an editor. The endpoint may have been created manually, or as an effect of creating a link earlier. The selected object does not have to be an endpoint.

   This operation is available from the editors and the Entity Dictionary. See "Link > Create" on page 442.

3. Automatically, by copying and pasting an object (*Paste As*).

   This operation does not require any existing endpoints. An object is first selected and copied in an editor. The object is then pasted in an editor or in the Organizer by using the *Paste As* menu choice. This transforms the object, if necessary, and automatically creates a link between the copied and pasted object.

   This operation is available from the SDL, MSC, OM and Text Editors and the Organizer and supports the SOMT method. See "The Paste As Command" on page 448. It is **not** available in the SC/HMSC Editor.

The Entity Dictionary

The Entity Dictionary Concept

The purpose of the Entity Dictionary is to provide easy access to names of entities being used in the system, and a possibility to reuse these names in all parts of the system. The entities that the Entity Dictionary manage are all the link endpoints defined in the system, as well as all diagrams, documents and modules found in the Organizer structure.

The Entity Dictionary is accessible from the SDL Suite editors. The names in the Entity Dictionary are available for reuse in all texts and graphical objects found in the graphical diagrams. However, the Text Editor does not support reuse of texts in text documents.

The Entity Dictionary is also used for creating links between objects and existing endpoints when using the editors.

Relations to Editor Windows

The Entity Dictionary is implemented as a modeless dialog window. There is not a single Entity Dictionary window, but one window for each type of diagram (SDL, MSC, OM, SC, HMSC and text). All Entity Dictionary windows contain exactly the same information, and all the windows are updated when a change is made. The reason for having an Entity Dictionary window for each editor type is that operations in the window apply to the object currently selected in the respective editor.

Since there might be several editor windows showing different diagrams/documents, there is a need to define the *current window*. This is the editor window that will be associated with the Entity Dictionary, and all operations will act on the current window. The current window is the editor window where the user last performed a menu command or a mouse click detected by the editor.

The editor associated with an Entity Dictionary window is known as the *parent editor*. The editor type is reflected in the window title of the Entity Dictionary window, thus making it possible to distinguish the different Entity Dictionary windows, and to determine the parent editor that the Entity Dictionary operations will affect.

Entity Dictionary Window

The Entity Dictionary Window can be opened from all of the editors through the use of the menu choice *Entity Dictionary* in the *Window* menu.

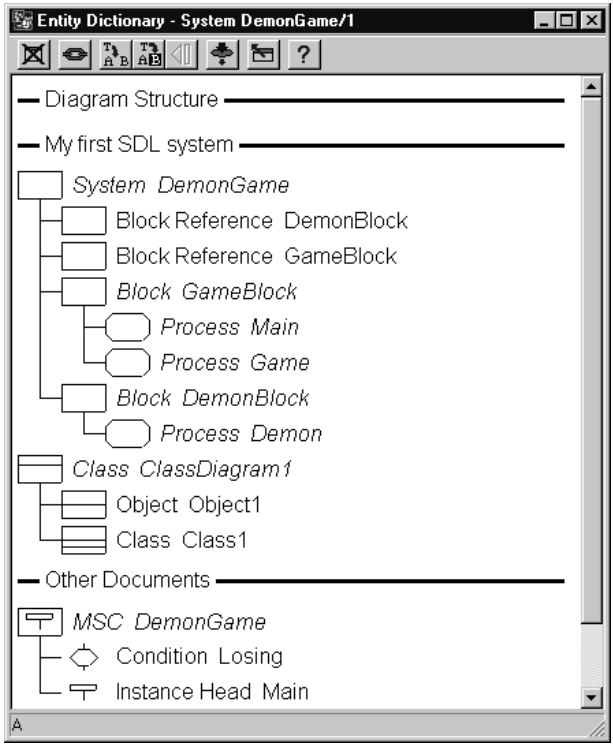


Figure 132: The Entity Dictionary window

Contents and Structure

The window lists all the defined link endpoints, following the structure of diagrams and files in the Organizer. By default, every item in the Organizer is repeated, and the indentation of items is also repeated. All Organizer items and link endpoints listed in the window are known as *entities* in the Entity Dictionary.

Below each Organizer item, the link endpoints defined in that item are listed with indentation:

- Below each SDL diagram, the symbols marked as link endpoints are listed.
- Below each MSC diagram, the instances, messages and other symbols marked as link endpoints are listed.
- Below each OM diagram, the names of all the classes, instances and other symbols marked as link endpoints are listed.
- Below each text document, the text fragments marked as link endpoints are listed.

For an Organizer item that contains both sub-documents and link endpoints, the endpoints are listed first, followed by the sub-documents.

It is possible to hide the Organizer items to display only the link endpoints; see "The Filter Dialog" on page 440 for more information.

The information in the Entity Dictionary is updated whenever a link endpoint is created, changed or cleared in any of the editors or the Link Manager, or when the Organizer structure is modified.

Entity Icons

Each entity has an associated icon that identifies the type of the entity, i.e. the type of the endpoint object. Entities that already have an established icon in the Organizer or the Index Viewer use the same icon in the Entity Dictionary, with a few exceptions. The icons specific to the Entity Dictionary are:


Diagram Heading


Diagram Extended Heading


OM Class


OM Object


OM Association


OM Aggregation


OM Generalization


SC Transition


MSC Message


MSC Create


MSC Timer Set


(H)MSC Reference


HMSC Start


HMSC Stop


HMSC Connection


Text symbols and text fragments

Textual Notation

The type and name of an entity is shown to the right of the icon. For entities containing a name, such as diagram symbols, this name is listed. For other symbols and text fragments, the first 25 characters are shown.

The cases when a diagram reference symbol, a diagram heading, or an Organizer document is an endpoint will result in duplication of information in the Entity Dictionary. To distinguish between such endpoints and the structure of Organizer documents, the following textual notations are used:

- Link endpoints are listed using a plain font face.
- Documents and other Organizer items that are not endpoints are listed using an *italic font.*
- Endpoints that are diagram reference symbols contain the word “Reference” after the diagram type.
- Endpoints that are diagram headings has the word “Heading”, “Additional Heading” or “Extended Heading” as the entity type.

Figure 132 on page 435 shows examples of these textual notations.

Operations in the Entity Dictionary

The Entity Dictionary window does not contain a menu bar. Operations are available as quick buttons or as popup menus.

Quick Buttons

The following quick buttons are special to Entity Dictionary window.

Close
Close the Entity Dictionary window.

Create Link
Create a link between the endpoint selected in the Entity Dictionary and the object selected in the parent editor. The quick button is dimmed if not two such selections are present. If the parent editor is a Text Editor, the selected text must be an already existing endpoint.
The *Create Link* dialog is opened; see Figure 135 on page 443.

Insert
Insert the text in the selected symbol in the Entity Dictionary at the insertion point in the object selected in the parent editor. The quick button is dimmed if not two such selections are present. If text is selected in the parent editor's text window, this text is instead replaced.
This button is not available in the Text Editor's Entity Dictionary.

Replace
Replace the text content of the object selected in the parent editor with the text in the selected symbol in the Entity Dictionary. The quick button is dimmed if not two such selections are present.
This button is not available in the Text Editor's Entity Dictionary.

Undo
Undo the most recent text operation in the Entity Dictionary (Insert, Replace, Undo). The quick button is dimmed if the selection in the parent editor has changed to another object.
This button is not available in the Text Editor's Entity Dictionary.

Filter
Filter the information listed in the Entity Dictionary. See "The Filter Dialog" on page 440 for more information.

Show Editor
Raise the parent editor window.

Popup Menus

The following tables lists the available operations in the popup menus of the Entity Dictionary window.

On the Window Background

Appears when no symbol is selected and the menu is invoked in an area not containing any symbols.

Expand All	Expands all collapsed symbols.
Collapse All	Collapses all symbols; only root symbols will be shown. Collapsed symbols are indicated with a small triangle directly below the symbol.
Show Editor	Raises the parent editor's window.

On a Document Symbol

Appears if an Organizer document symbol is selected, or the menu is invoked where a document symbol is selectable.

Expand	Expands a collapsed symbol one level down.
Expand Substructure	Expands the entire substructure of the symbol.
Collapse	Collapses the substructure of the symbol.

On an Endpoint Symbol

Appears if an endpoint symbol is selected, or the menu is invoked where an endpoint symbol is selectable.

Show Definition	Brings up an editor window, or the Organizer window, where the endpoint is selected.

Double-Clicks

If there is a selection in the parent editor and a symbol in the Entity Dictionary is double-clicked, the text of the symbol is inserted in the parent editor. A double-click thus corresponds to using the *Insert* quick button; see "Insert" on page 438. This functionality is not available in the Text Editor's Entity Dictionary.

The Filter Dialog

The Filter dialog is opened when the *Filter* quick button is pressed in the Entity Dictionary window. The Filter dialog controls what is to be shown in the Entity Dictionary window.

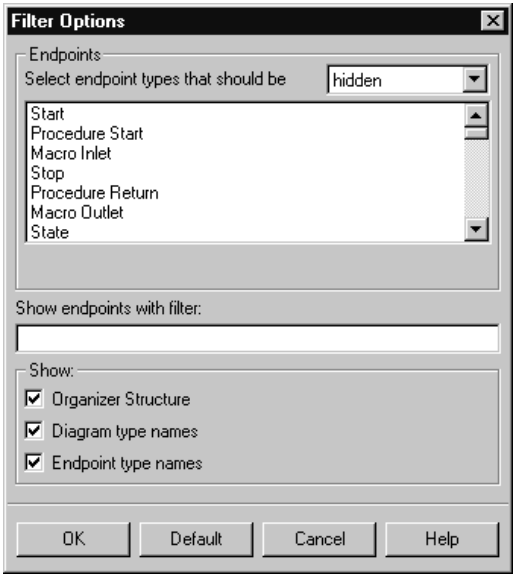


Figure 133: The Filter dialog

- *Select endpoint types that should be*

  This option menu controls whether the selected endpoint types should be *hidden* or *shown*. The possible endpoint types are shown in a multiple selection list, in which any number of items can be selected. If the option menu is set to *shown*, only the selected endpoint types are listed in the Entity Dictionary. If the option menu is set to *hidden*, the selected endpoint types are hidden, and the ones not being selected are thereby shown.

- *Show endpoints with filter*

  This text field is a pattern for matching endpoint names. Only endpoints whose names match the pattern are shown in the Entity Dictionary. The string has the same syntax as a normal UNIX file pattern, and may contain the elements '*' (zero or more characters), '?' (exactly one character), and '[...]' (any character within the brackets).

  An empty text field matches any name, i.e. it is equal to a single '*'.

- *Organizer Structure*

  This option controls whether the documents making up the Organizer structure are shown. If not set, only endpoints are listed in the Entity Dictionary.

- *Diagram type name*

  This option controls whether the diagram type names in the Organizer structure are shown. If not set, only the names of the diagrams are shown in the Entity Dictionary.

- *Endpoint type names*

  This option controls whether the endpoint type names are shown. If not set, only the names of the endpoints are shown in the Entity Dictionary.

- *Default*

  Resets the Filter dialog to its default settings, but does not close the dialog. The default settings are:

  – The endpoint type list contains no selection.
  – The endpoint type option menu is set to *hidden*.
  – The endpoint name filter is empty.
  – The three Show options are set.

Link Commands in the *Tools* Menus

This section describes the link-related commands that are available in the *Link* submenu in the *Tools* menu of the Organizer and all SDL Suite editors.

> **Note:**
>
> None of these commands are possible to *Undo*.
>
> The Organizer only supports a subset of the link commands.

The *Link* submenu contains the following menu choices:

- *Link > Create*
- *Link > Create Endpoint*
- *Link > Traverse*
- *Link > Link Manager*
- *Link > Clear*
- *Link > Clear Endpoint*

Link > Create

This menu command creates a link between the object selected in the editor and the object selected in the Entity Dictionary. If two such objects are selected, the Create Link dialog is opened (see Figure 135 on page 443). The editor object does not need to have a link endpoint defined to be able to create a link, i.e. only one of the objects need to be present in the Entity Dictionary. It is possible to create a link to itself.

> **Note:**
>
> In the Text Editor, the selected text must already be a link endpoint. It is not possible to create links to text that is not an endpoint.

If no object is selected in the editor, or no endpoint is selected in the Text Editor, the menu command is dimmed.

If no object is selected in the Entity Dictionary, or the Entity Dictionary has not been opened, a warning dialog is issued (see Figure 134) and the Entity Dictionary window is opened or raised. When the dialog is closed and the instructions given in it are followed, the Create Link dialog is opened.

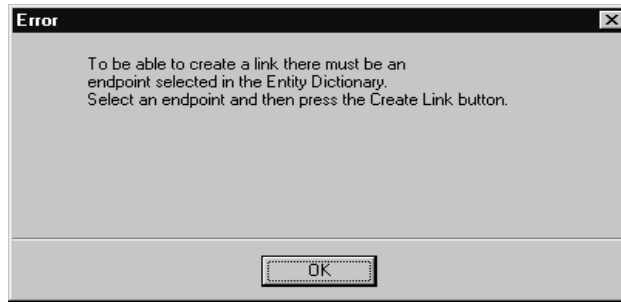


Figure 134: The Create Link warning

The Create Link dialog looks like this:

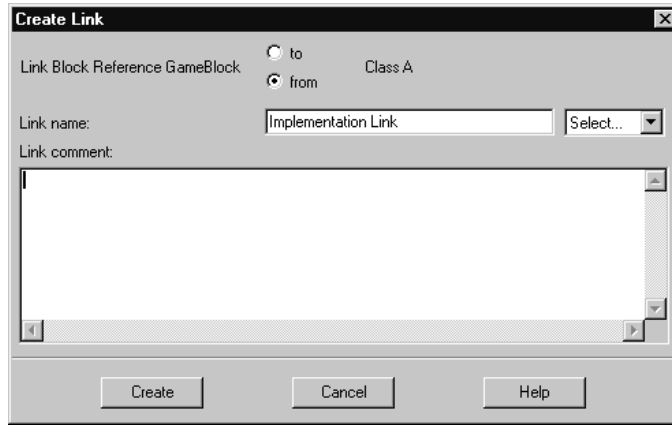


Figure 135: The Create Link dialog

- *Link <editor object> to/from <entity dictionary object>*

  The two selected objects are listed at either side of the *to/from* radio buttons. The radio buttons control which of the objects that is to be the logical to and from object. *Link to* is the default.

- *Link name*

  The name of the link. A name can be entered or edited in the text field, or be selected from the associated option menu. The five latest

used link names when creating links will be available in the option menu, and the name in the text field is preset to the latest used link name. A link name **must** be specified.

- *Link comment*

  An optional comment text, to be provided by the user. The text box is initially empty.

- *Create*

  Creates a link between the two objects. The link will be visible in the Link Manager and the endpoint objects are marked as being an endpoint with at least one link, i.e. if the editor object was not already an endpoint, it will be created.

If the Entity Dictionary window was opened because of the Create Link command, it will stay up until the user explicitly closes it.

Link > Create Endpoint

This menu command defines the currently selected object as a link endpoint. The endpoint is immediately added to the Entity Dictionary and the Link Manager. The object is marked as a link endpoint in the invoking tool – an editor or the Organizer. (See "Visualization of Endpoints and Links" on page 430.)

This command is dimmed if not exactly one object that can be defined as a link endpoint is selected, or if the object already is an endpoint.

This command is also available in the popup menus of all editors and the Organizer.

Link > Traverse

This menu command traverses an existing link from the currently selected object. This is done by opening the tool where the other link endpoint is defined (an editor or the Organizer) and selecting the other endpoint object.

This command is dimmed if more than one object is selected, or if the selected object has no links defined.

This command is also available in the popup menus of all editors and the Organizer, as *Traverse Link*.

If there is only one link defined from the object, that link is traversed, as described above. If more than one link exists, a dialog showing all links is opened and one of the links has to be selected.

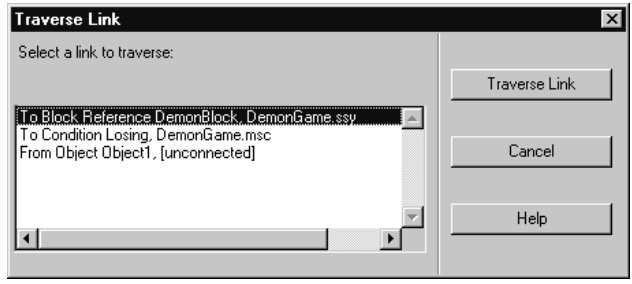


Figure 136: The Traverse Link dialog

The dialog presents a list of all links to and from the object in the following form:

`From` | `To` <*type and name of linked object*>, <*file name*>

The type and name of the linked object follow the same notation as in the Entity Dictionary; see "Textual Notation" on page 437. The file name of the document where the linked object is found contains a path if the file is not in the Organizer's Source Directory.

- To traverse a link, select the link and click the *Traverse Link* button, or double-click the link.

Link > Link Manager

This menu command opens or raises the Link Manager window. If exactly one object is selected and this object is a link endpoint, this endpoint will be selected and made visible in the Link Manager. This command is never dimmed.

For more information about the Link Manager, see "The Link Manager" on page 462.

Link > Clear

This menu command removes one or more links to or from the currently selected object. However, the link endpoints are preserved. This command is dimmed if more than one object is selected, or if the selected object has no links defined.

Caution!

Removing a link cannot be undone.

A dialog showing all links is opened and one or more of the links has to be selected.

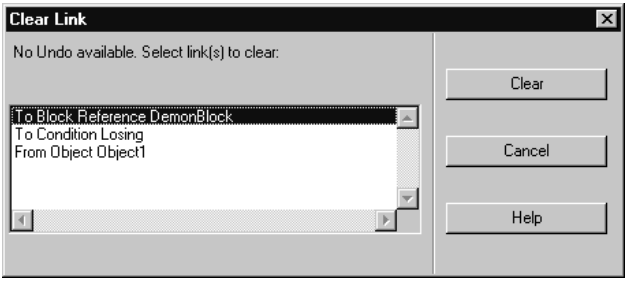


Figure 137: The Clear Link dialog

The links to and from the object are listed in the following form:

`From | To` <*type and name of linked object*>

The list of links is a multiple selection list, in which one or more links can be selected. When a selection is made in the list, the *Clear* button becomes active.

To remove links, select the links in the list and click the *Clear* button. A link is removed from both endpoint objects, but the link endpoints themselves are **not** removed. If either of the endpoint objects had only this link defined, and no others, the mark of the object changes to indicate that the object is only an endpoint with no links defined. For information on how to remove an endpoint, see "Link > Clear Endpoint" on page 447 (below).

Link > Clear Endpoint

This menu command removes the link endpoint from the currently selected object, and subsequently all links connected to the endpoint.

This command is dimmed if more than one object is selected, or if the selected object is not an endpoint.

If the selected object has no links connected to the endpoint, the endpoint is removed without further user interaction. The endpoint is removed from the Entity Dictionary and the Link Manager. The object is no longer marked as a link endpoint in the invoking tool (an editor or the Organizer).

Note:

In the Text Editor, this command can also be used to reduce the extent of an already existing endpoint. If the selection only indicates a partial range of the endpoint, at the start or end of the endpoint, a dialog will appear allowing you to choose whether to remove the entire endpoint, or just remove the selected part of the endpoint from the selection.

If a partial range in the middle of the endpoint text is selected, the only possibilities are to remove the entire endpoint or cancel the operation.

If the selected object has one or more links connected to the endpoint, a warning dialog is opened, since the operation of removing a link cannot be undone:

The *Paste As* Command

The command *Paste As* is available in the *Edit* menu of the Organizer and in the SDL Suite editors (except for SC or HMSC diagrams).

The Paste As command is used to paste a copied object as another object, and at the same time create an implementation link between the copied and pasted objects. It is also possible to paste a cut object, but in this case no link can be created.

It is possible to paste the object into an editor different from the one the object was copied from. This requires a transformation of the object according to the user's choice.

Paste As supports the SOMT method, which governs the possible transformations for a particular object. See the SOMT Methodology Guidelines starting in chapter 69 in the User's Manual for information and advice on when to use a particular transformation.

> **Note:**
>
> The normal *Paste* command in the *Edit* menu is very different from *Paste As*. A normal paste can only be performed in the same editor as the object was copied from, and the pasted object is as far as possible an identical copy of the object.

The Paste As Process

The process of using Paste As consists of the following steps:

1. Copy (or cut) an object to the clipboard.

   A **single** object is selected and copied to the clipboard by using the *Copy* command in the *Edit* menu of the editor. The Paste As command supports the following objects being copied:

   – A class symbol in an OM Editor.
   – An object symbol in an OM Editor.
   – A text fragment in a Text Editor that either contains no endpoints or exactly matches an endpoint.

   It is thus not possible to use Paste As with copied SDL or MSC symbols, or with Text symbols copied in an OM Editor.

2. Paste the object using *Paste As*.

   In the desired editor or the Organizer, the *Paste As* command is selected from the *Edit* menu. The menu choice is dimmed if:

   – More than one object was copied.

   – An object different from the list above was copied.

   – The copied object cannot be pasted into the tool, i.e. there is no transformation defined for this particular object–tool combination. The possible transformations are listed in "Transformation Scheme" on page 452.

   After selecting the menu choice, the Paste As dialog is opened. See "The Paste As Dialog" on page 450.

3. Select the type of object to paste the copied object as.

   In the Paste As dialog, the possible resulting object types are listed in an option menu. The object types listed reflects the transformations possible for this particular copy–paste situation. If the desired object type is not present in the list, the user may have to change which object is being copied, the editor where the paste is made, or (for the SDL Editor) the type of SDL diagram being pasted into.

4. Select the type of link to create, if any.

   In the Paste As dialog, it is possible to change the default of creating an "Implementation Link" between the copied and pasted objects. See "The Link Info Dialog" on page 451. If the object was cut instead of copied, no link can be created.

5. Place the pasted object.

   If the pasted object is a graphical symbol, a "floating" symbol must be placed with the mouse in the usual way, and the paste can be cancelled by pressing `<Esc>`. If the pasted object is a textual description or an Organizer document, the object is placed at the text cursor or the current selection.

6. Edit the pasted object, if needed.

   The results of the object transformation may not be complete or accurate. The user may need to change the pasted object to achieve the desired result.

The *Paste As* Dialog

The Paste As dialog is opened when *Paste As* is selected from the *Edit* menu.

Figure 138: The Paste As dialog

- *Paste <copied object> as*

  The option menu contains all possible types of objects that can be created in the current situation; in some cases, only a single alternative is available. A default object type is pre-selected. The possible object types and the default are presented in "Transformation Scheme" on page 452.

- *Create link from copied object to pasted object*

  This option controls whether a link is to be created between the copied and pasted objects. This is by default set, and the link will always be made from the copied object to the pasted object. If the object was cut instead of copied, this option is dimmed.

- *Link name: <link name>*

  States the name of the link to create, and is only valid if the *Create link* option is set. By default the link name is "Implementation Link", but this can be changed in the Link Info dialog; see below.

- *Paste As*

  Closes the dialog, creates an object of the selected type, pastes it in the invoking tool (the Organizer or an editor), and optionally creates a link. In graphical editors, pressing `<Esc>` cancels the paste.

- *Link Info*

  Brings up the *Link Info* dialog (see below), where the attributes of the link can be changed before it is created. This button is dimmed if the *Create link* option is dimmed or not set.

The *Link Info* Dialog

The *Link Info* dialog is opened when the *Link Info* button is pressed in the *Paste As* dialog.

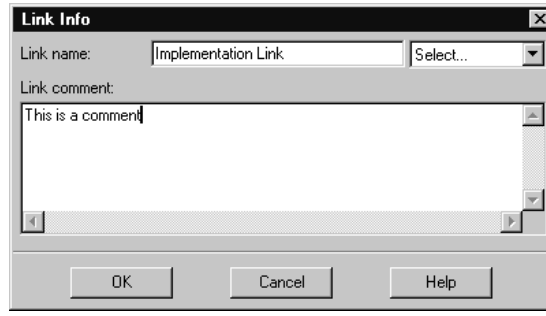


Figure 139: The Link Info dialog

- *Link name*

  An editable text field specifying the name (type) of the link to create. The name is preset to “Implementation Link”. The five latest used link names when creating links is available in the *Select* option menu. Selecting a name from this menu inserts the name into the text field.

  The link name text field must not be empty.

- *Link comment*

  An optional comment text to be provided by the user. The text in the comment field is initially empty. The comment associated with a link can only be viewed and changed later on by using the Link Manager.

Transformation Scheme

The table below presents all the possible object type combinations for the copied and pasted object in the Paste As operation. The preselected choice shown when the Paste As dialog is opened is shown in **bold face.**

Copied object	Paste As in OM Editor	Paste As in SDL Editor	Paste As in MSC Editor	Paste As in Text Editor	Paste As in Organizer
Class (in OM Editor)	**Class** Object	System Type Block Type Block **Process Type** Process Service Type Service Text symbol with NEWTYPE Text symbol with SDL interface	—	**C++ class** C struct IDL Module IDL interface ASN.1 sequence	**System**
Object (in OM Editor)	**Class** Object	Block instance **Process instance** Service instance	—	—	**System**
Text fragment (in Text Editor)	**Class** Object	—	**Instance** Message	**Text fragment**	**MSC**

Note:

Not all object types are possible to Paste As in all situations. Especially in the SDL Editor, the available object types depend on which diagram type the Paste As is performed in.

Transformation Details

The details of the specific object transformations are described in the following subsections. Please refer to the above table to see which tools that support a particular object transformation.

Some general transformation details are:

- The name of a copied symbol can be empty. When pasted as an SDL diagram, the name “EmptyName” will be used.
- In SDL diagrams, if the name used for a pasted symbol will be in conflict with an already existing name, the pasted name will be the original name suffixed by “_<number>”. For example, if “Name” already exists for a reference symbol in an SDL diagram, the pasted symbol will have the name “Name_1”. The number is incremented until the name is unique.
- The size of the pasted SDL, MSC and OM symbols are the same as when a symbol is manually picked from the editor’s symbol menu.
- The exact layout of generated diagrams may not be depicted correctly in the following illustrations. Only the upper left part of a diagram is shown.

Pasting an OM Class

An OM class may be represented and defined by several class symbols in the OM diagrams. This is the case if more than one class symbol with exactly the same class name is found within the OM scope (the diagram itself or the diagrams in the same Organizer module).

When an OM class is pasted, **all class symbols defining the class in the scope are considered**. It is the combined set of attributes and operations in the class symbols that will be used in the transformation to the pasted object.

In the following descriptions of object transformations, we use the two class symbols shown in Figure 140 (within the same scope) as a generic example. Regardless of which class symbol is copied, both class symbols are considered in the transformations.

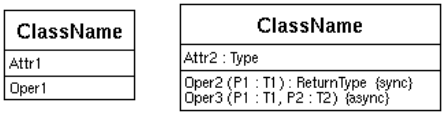
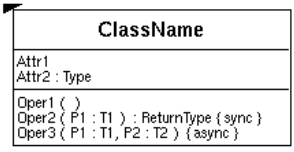
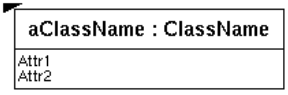
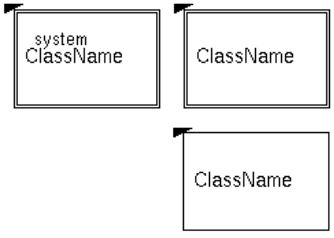
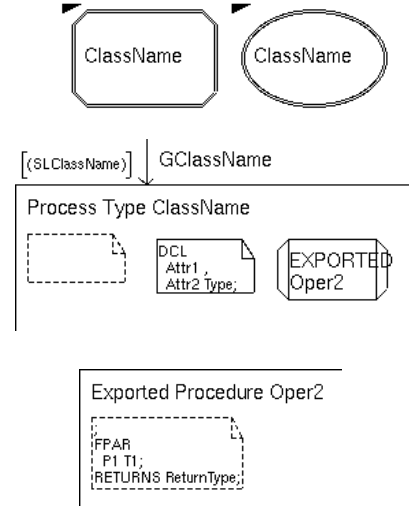


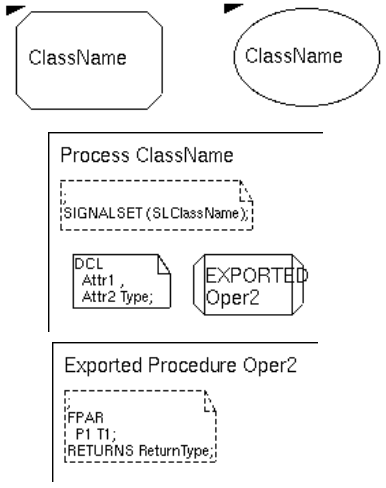
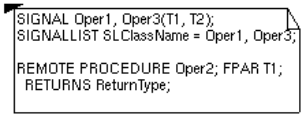
Figure 140: Two class symbols in the same scope

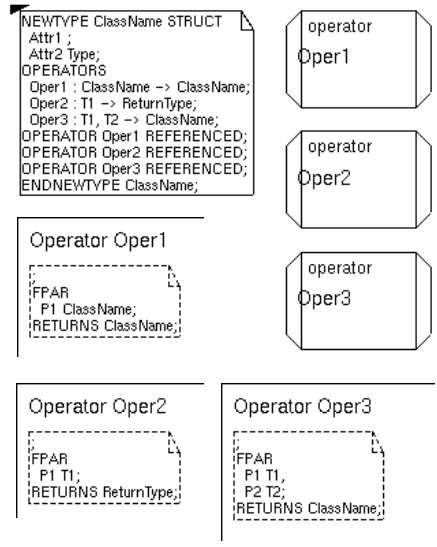
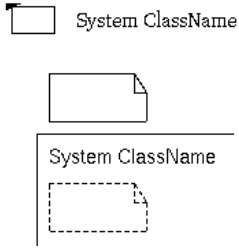
Note:

Even if there are more than one class symbol in the scope, only the copied symbol will be linked with the pasted object (if a link is created). Links are not created for the other symbols in the scope.

Paste As object, with description	Resulting objects and diagrams
Class symbol The pasted class is simply a copy of the merged class.	
Object symbol The pasted object contains the merged attributes of the class. The object is named ‘a’ followed by the class name.	
System Type diagram **Block Type diagram** **Block diagram** In these cases, only a reference symbol is created, not any contents of the diagram.	

Paste As object, with description	Resulting objects and diagrams
Process Type diagram **Service Type diagram** A reference symbol is created, as well as the referenced diagram with some contents. The keyword “{async}” after an operation means that it will be transformed using a signal interface. A gate named ‘G’ followed by the class name, and a signal list named “SL” followed by the class name, is added. The keyword “{sync}” after an operation means that it will be transformed using an RPC interface. The procedure diagram is created with an additional heading symbol containing FPAR and RETURNS statements for parameters and return type. If no keyword is given, signal interface is the default for operations without return value, and RPC interface is the default for operations with return value. **NOTE:** Text symbols containing the declarations of the signal list and the remote procedures must be created by a separate Paste As operation, usually in a diagram at a higher level. See “Text symbol with SDL interface” on page 456 (below).	

Paste As object, with description	Resulting objects and diagrams
Process diagram **Service diagram** The same transformation as for process type and service type, but the signal interface is added as a SIGNALSET statement in the additional heading symbol. **NOTE:** Text symbols containing the declarations of the signal list and the remote procedures must be created by a separate Paste As operation, usually in a diagram at a higher level. See "Text symbol with SDL interface" on page 456 (below).	
Text symbol with SDL interface A text symbol is added, containing declarations for the signal list and/or remote procedures, as described for process type and service type above.	

Paste As object, with description	Resulting objects and diagrams
Text symbol with NEWTYPE A text symbol is added, containing a NEWTYPE definition. Operator diagrams are added for all operations, with additional heading symbols containing FPAR and RETURNS statements for parameters and return type. If an operation does not have parameters, a parameter is inserted with the class name as type. The class name is also used as return type if an operation does not specify any.	
System diagram In the Organizer, the system diagram is added in the same way as for the *Add New* operation, i.e. the diagram is added as a new root diagram at the current selection. The diagram is also opened in the SDL Editor.	
C++ class definition The link to the class definition is inserted inside a C++ comment.	`// class ClassName` `class ClassName {` `public:` `  void Oper1();` `  ReturnType Oper2(T1 P1);` `  void Oper3(T1 P1, T2 P2);` `private:` `  Attr1;` `  Type Attr2;` `};`

Paste As object, with description	Resulting objects and diagrams
C struct definition The link to the struct definition is inserted inside a C comment.	`/* struct ClassName */` `typedef struct {` `  Attr1;` `  Type Attr2;` `} ClassName;` `void Oper1(ClassName *);` `ReturnType Oper2(ClassName *,` `  T1 P1);` `void Oper3(ClassName *,` `  T1 P1, T2 P2);`
IDL module Only the class name is used. The link to the module is inserted inside an IDL comment.	`// module ClassName` `module ClassName {` `};`
IDL interface Operations marked with keyword "{async}" will get the string `oneway void` inserted before the name of the operation. The link to the interface is inserted inside an IDL comment.	`// interface ClassName` `interface ClassName {` `  attribute Attr1;` `  attribute Type Attr2;` `  oneway void Oper1();` `  ReturnType Oper2(T1 P1);` `  oneway void Oper3(T1 P1,` `  T2 P2);` `};`
ASN.1 sequence The link to the sequence is inserted inside an ASN.1 comment.	`// ClassName SEQUENCE` `ClassName ::= SEQUENCE {` `  Attr1,` `  Attr2 Type` `}`

Pasting an OM Object

In a similar way as when pasting an OM class, all object symbols of exactly the same class in the scope are considered. It is the combined set of attributes in the object symbols that will be used in the transformation to the pasted object.

In addition, the class that the object is an instance of is also considered, if it exists. That is, it is the combined set of attributes and operations in the class symbols and the objects symbols that will be used in the transformation.

In the following descriptions of object transformations, we use the object symbol shown in Figure 141 as a generic example. In addition, the object is assumed to be located in the same scope as the two class symbols shown in Figure 140 on page 454.

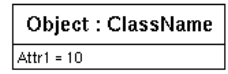
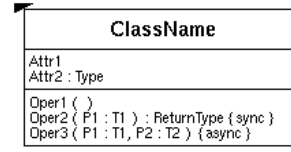
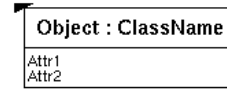
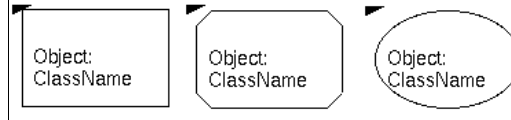


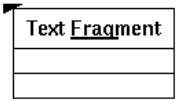
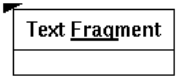
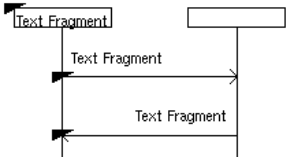
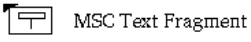
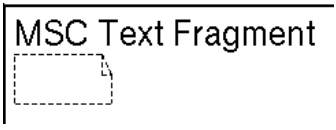
Figure 141: An Object symbol

Paste As object, with description	Resulting objects and diagrams
Class symbol The pasted class uses the class name, attributes and operations from the copied object and the object's class symbols.	
Object symbol The pasted object uses the name and attributes from the copied object and the object's class symbols, but without attribute values.	
Block instance diagram **Process instance diagram** **Service instance diagram** The pasted diagram becomes an instance diagram (indicated in the Organizer).	
System diagram Works in the same way as when pasting an OM class. The name of the diagram will be "Object : ClassName".	See "System diagram" on page 457.

Pasting a Text Fragment

In the following descriptions of object transformations, we use the text fragment “Text Fragment” as a generic example.

> **Note:**
>
> The copied text fragment must either contain no endpoints, or exactly match an existing endpoint in the text. If the text fragment contains both endpoint text and non-endpoint text, it cannot be used for Paste As.

Paste As object, with description	Resulting objects and diagrams
Class symbol Syntax check is performed on the class name.	
Object symbol Syntax check is performed on the object name.	
Text fragment The pasted text is simply a copy of the copied text.	Text Fragment
MSC instance **MSC message out** **MSC message in** The two message types place the endpoint at different ends of the message line.	
MSC diagram In the Organizer, the MSC diagram is added in the same way as for the *Add New* operation, i.e. the diagram is added as a new root diagram at the current selection. The diagram is also opened in the MSC Editor.	 

Other *Edit* Commands

Some of the commands in the *Edit* menu in the editors are affected when operating on objects that have endpoints with connected links.

Pasting an Object

If objects with endpoints are cut or copied, the endpoints and any existing links to the objects are saved in the clipboard.

When pasting (by using the ordinary *Paste* command) an object with an endpoint, but without connected links, the endpoint is pasted together with the object without further user interaction.

When pasting objects that also have links connected to their endpoints, the following dialog appears:

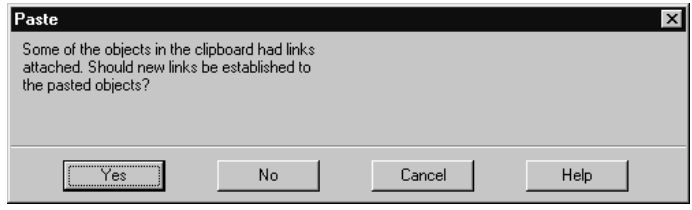


Figure 142: The Paste dialog

- The *Yes* button pastes the objects and keeps the links. An object having connected links will be pasted with new links created between the pasted object and the objects the original object was linked to.
- The *No* button pastes the objects without endpoints and links. No pasted objects will have any endpoints or connected links.

Deleting an Object

If you delete an object that has links, the link information will be destroyed and cannot be restored. Therefore, when you want to cut or clear objects with links, a warning dialog will be issued where it is possible to cancel the operation.

The Link Manager

The Link Manager manages endpoints and links in a system. The Link Manager handles the link file, containing information about the endpoints and links in a system. The syntax of the link file is described in "The Link File" on page 485.

For an overview of link concepts, see "Link Concepts and Overview" on page 428. In addition, the following concepts are used in the Link Manager:

- *Entity*

  A collection of endpoints with the same type and name in the same scope is called an *entity*. The scope is defined as the file the endpoint resides in. If the file is in an Organizer module, the scope is the module.

- *Cardinality*

  The number of links associated with an endpoint, i.e. the total number of links going to and from the endpoint.

Link Manager Window

The Link Manager's window is shown in Figure 143. The window title contains the name of the loaded link file, and is appended by an asterisk '*' if the link file is modified.

In the window, the endpoints in the system are presented graphically using icons. The links between the endpoints are represented as lines between them, with an arrow stating the direction of the link. The Link Manager can present different *views* of the endpoint/link information; these are described in "Presentation Views and Link Trees" on page 464.

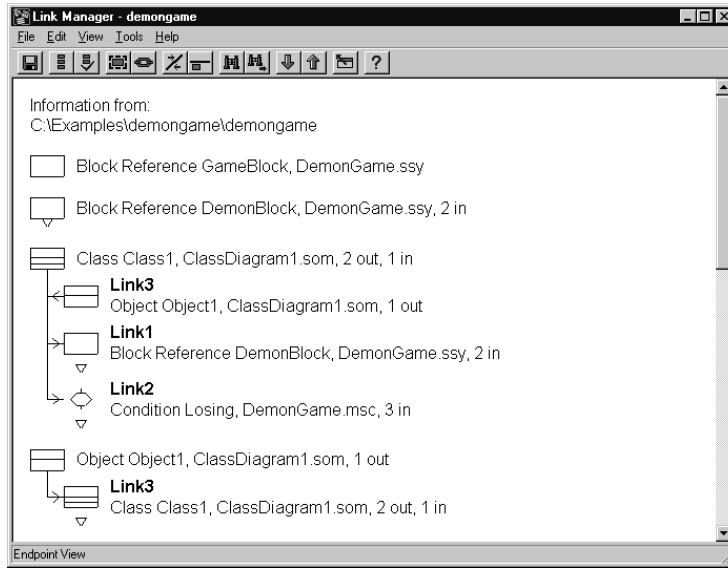


Figure 143: The Link Manager window

At the top of the drawing area, the name of the current link file is presented under a “Information from” heading. If several link files are merged, they are all listed here until the next *Save* operation. If a local link file is used (see “Local Link File” on page 429), both the master link file (read only) and the local link file are listed.

If the information in the Link Manager is not yet saved, “on-line editing” will be displayed to indicate that endpoints have been created in an editor or in the Organizer.

The main part of the drawing area displays all endpoints and links as *link trees*. Every endpoint is a root node, and the associated links and endpoints are added to that root. Link trees can be collapsed, and endpoints and links can be hidden.

Below the link trees, some statistics are presented under a “Statistics” heading. The number of endpoints and links is displayed, including the number of hidden endpoints and links, as in the following example:

```
Statistics:
10 endpoints (2 not shown)
```

```
4 links (1 not shown)
```

Endpoint Icons

The Link Manager uses the same icons for endpoints as the Entity Dictionary. See "Entity Icons" on page 436 for more information.

The icons can have different layouts indicating the state of the associated endpoint:

Normal
The normal state of the icon. Information is not modified.

Invalid
An endpoint is marked invalid if, after a *Check Endpoints* operation, it is not present in the document where it was supposed to be.

Dirty
The endpoint is modified or newly created from an editor, but the link file is not yet saved.

Dashed
The endpoint is already displayed on a higher level in the same tree, or it belongs to the *TO* group after a *Consistency Check*.

The endpoints are added as new root symbols below the last link tree as they are created. The order of the icons can be changed by using the quick buttons *Move Down* and *Move Up*.

Presentation Views and Link Trees

The Link Manager has two main methods of displaying endpoints, using an *Endpoint view* or an *Entity view*. There is also a *Consistency view*, used for presenting the results after a consistency check has been performed (see "Consistency Check" on page 476). When there is no selection in the drawing area, the name of the currently displayed view is shown in the status bar.

In Endpoint view, a link tree looks like this:

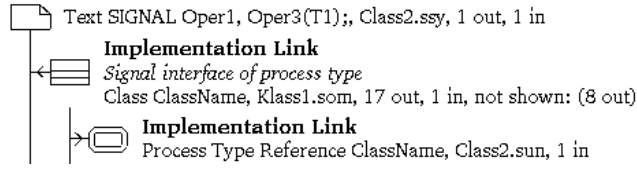


Figure 144: Link tree in Endpoint view

To the right of each endpoint icon, the following information identifying the endpoint is displayed (depending on the options set in *Options > Endpoint*):

- The endpoint type, i.e. the type of the endpoint object.
- The name of the endpoint, in a plain type face. For endpoints containing a name, such as diagram symbols, this name is listed. For other symbols and text fragments, the first 25 characters are shown.
- The file the endpoint resides in. The file name is shown with or without its absolute path depending on the setting in the Organizer.
- The link cardinality; the number of “out” links followed by the number of “in” links.
- The number of hidden links, if any, within parenthesis.

For each link in a link tree, the name of the link is displayed above the endpoint information, in **bold face**. The link name display can be switched on or off by the menu choice *Options > Link*.

The link comment, if it is used, is shown directly below the link name, in *italics*. The comment display can be switched on or off by the menu choice *Options > Link*.

In Entity view, all endpoints representing the same entity are collected into one symbol, and the number of endpoints represented by that entity is presented to the right of the name of the entity, preceded by an asterisk ‘*’. Also, instead of displaying the filename the endpoint resides in, the scope the entity resides in is displayed. The scope is either a filename or a module. An example of an endpoint in Entity view:

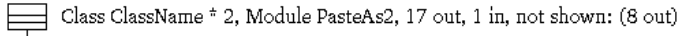


Figure 145: An entity in Entity view

Link Manager operations on endpoints also apply to entities. If there is a difference in the behavior of a menu choice depending on the view, this will be pointed out in the descriptions of the operations.

Double-Clicks

Double-clicking on an icon invokes the menu choice *Show in Editor*. In Entity view, if the selected entity corresponds to several endpoints, they will be selected one at a time for each double-click.

Menu Bar

This section describes the menu bar of the Link Manager window and all the available menu choices.

The menu bar contains the following menus:

- *File Menu*
- *Edit Menu*
- *View Menu*
- *Tools Menu*
- *Help Menu*
  (see "Help Menu" on page 15 in chapter 1, *User Interface and Basic Operations*).

File Menu

The *File* menu contains the following menu choices:

- *New*
- *Open*
- *Merge*
- *Save*
- *Save As*
- *Print*
- *Close*

The menu choices are described in "File Menu" on page 8 in chapter 1, *User Interface and Basic Operations*, except *Print*, which is described in "The Print Dialogs in the SDL Suite and in the Organizer" on page 316 in chapter 5, *Printing Documents and Diagrams*, and *Merge*, which is described below.

Merge

This menu choice opens an existing link file, and merges the contents of that file with the information already in the Link Manager. It works in a similar way to *Open*, but keeps the current endpoint and link information.

If two links are equal (i.e. they have the same source and destination endpoints and the same name) but they have different link comments, the new comment will consist of the old comments separated by a newline character.

Edit Menu

The *Edit* menu contains the following menu choices:

- *Highlight Endpoint*
- *Replace Endpoint*
- *Create Link*
- *Link Details*
- *Clear Link*.

Highlight Endpoint

This menu choice highlights an endpoint. Highlighting an endpoint is the first step to replace an endpoint or create a link. The highlighting is presented as a frame around the highlighted endpoint.

The first time this menu choice is used, the selected endpoint will be highlighted. The second time this menu choice is used for the same endpoint the highlighting will be removed. There is at most one highlighted endpoint. If another endpoint already was highlighted, the highlighting is moved to the selected endpoint.

Replace Endpoint

This menu choice replaces an endpoint with another endpoint. This operation is useful if an endpoint has become Invalid and the user has found a replacement endpoint that all links should be moved to.

One endpoint is defined with the *Highlight Endpoint* menu choice, the other endpoint is defined by the selection.

All links going to or from the replaced endpoint will be updated to go to or from the other endpoint instead. If the replaced endpoint was invalid, the user is given the option to delete the replaced endpoint.

The following dialog appears:

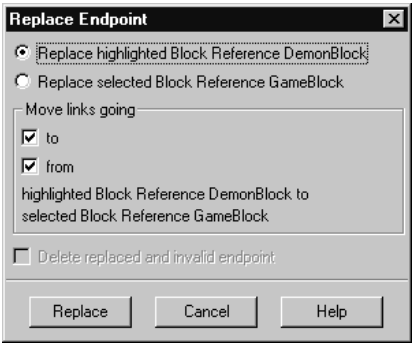


Figure 146: The Replace Endpoint dialog

- *Replace highlighted* `<endpoint>`
- *Replace selected* `<endpoint>`

  Depending on the setting of the radio button, either the highlighted or the selected endpoint will be replaced.

- *Move links going to/from*

  These options are used to select if links going to the replaced endpoint, or links going from the replaced endpoint, will be moved.

- *Delete replaced and invalid endpoint*

  If the endpoint to be replaced is invalid, it can optionally be deleted. If any links going to the deleted endpoint are not moved, they will also be deleted.

Create Link

This menu choice creates a link between the highlighted endpoint and the selected endpoint.

One endpoint is defined with the *Highlight Endpoint* menu choice, the other is defined by the selection.

The Create Link dialog appears, see Figure 135 on page 443.

Link Details

This menu choice displays information about the link above the selected endpoint; the name, the comment, and the direction of the link. All these attributes can be edited.

The following dialog appears:

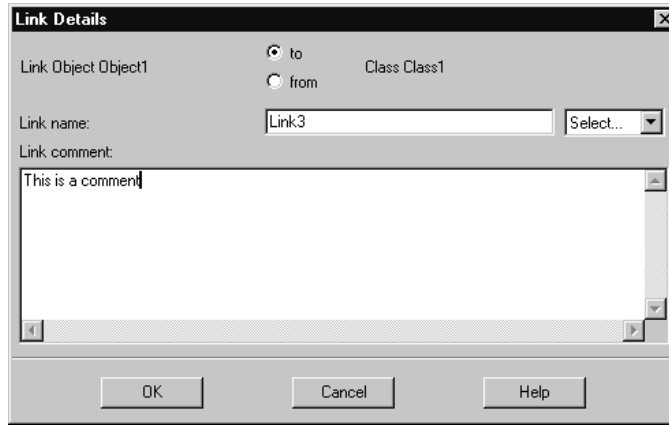


Figure 147: The Link Details dialog

The dialog works in the same way as the Create Link dialog, see Figure 135 on page 443.

Clear Link

This menu choice clears (deletes) the link above the selected endpoint. Only the link will be cleared, not the associated endpoints. You will be asked to confirm or cancel the deletion.

View Menu

The *View* menu contains the following menu choices:

- *Expand*
- *Expand Substructure*
- *Collapse*
- *Options > Window*
- *Options > Link*
- *Options > Endpoint*
- *Filter*
- *Set Scale*.

Expand

This menu choice expands the endpoint structure tree one level down for the selected endpoint. If any endpoints one level down are hidden, they will still be hidden after this operation. (Use the *Filter* menu choice to show or hide endpoints).

The menu choice is dimmed if:

- No endpoint is selected
- The selected icon is a leaf (no children icons)
- The selected icon is already expanded

Expand Substructure

This menu choice expands the endpoint structure tree the whole way down for the selected endpoint. If there is no selection, all endpoint trees will be expanded.

Collapse

This menu choice collapses the selected endpoint, i.e. the sub symbols are not shown after this operation. A collapsed endpoint has a small triangle drawn below the icon to indicate that it is collapsed. If there is no selection, everything will be collapsed.

Options > Window

This menu choice sets options for controlling the appearance of the Link Manager window.

The following dialog appears:

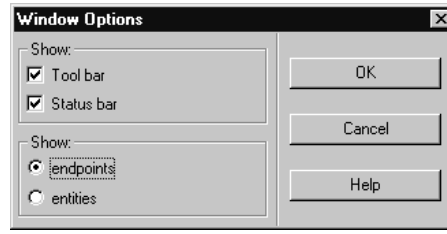


Figure 148: The Window Options dialog

- *Show: Tool Bar*
- *Show: Status Bar*

  These options control whether the tool bar and the status bar should be displayed or not.

- *Show: endpoints/entities*

  By using this radio button, Endpoint or Entity view is selected (see "Presentation Views and Link Trees" on page 464).

Options > Link

This menu choice sets options for controlling the appearance of links in the drawing area.

The following dialog appears:

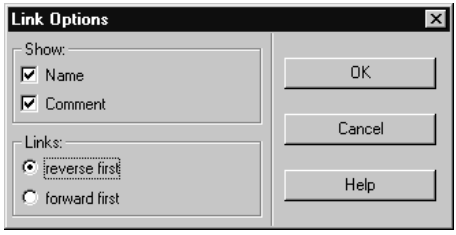


Figure 149: The Link Options dialog

- *Show: Name*

  Show/hide the name of all links.

- *Show: Comment*

  Show/hide the comment for all links.

- *Links: reverse first/forward first*

  This setting controls whether links going to a root endpoint (*reverse first*) or links going from a root endpoint (*forward first*) will be displayed first in the link trees.

Options > Endpoint

This menu choice sets options for controlling the appearance of endpoints in the drawing area.

The following dialog appears:

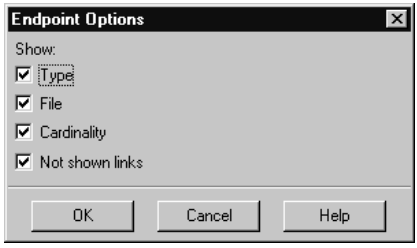


Figure 150: The Endpoint Options dialog

- *Show: Type*

  Show/hide the type of the endpoints.

- *Show: File*

  Show/hide the file name (or module) of the endpoints.

- *Show: Cardinality*

  Show/hide the cardinality, i.e. the number of links going to and from an endpoint.

- *Show: Not shown links*

  Show/hide the number of hidden links going to and from an endpoint.

Filter

This menu choice is used for filtering out endpoints and/or links which will not be shown. The filter is set in a modeless dialog, i.e. the Link Manager continues working without waiting for the dialog to be closed.

If an endpoint is hidden, all links associated to it will be hidden. If a link is hidden, the endpoints associated to it will still be visible.

The following dialog appears:

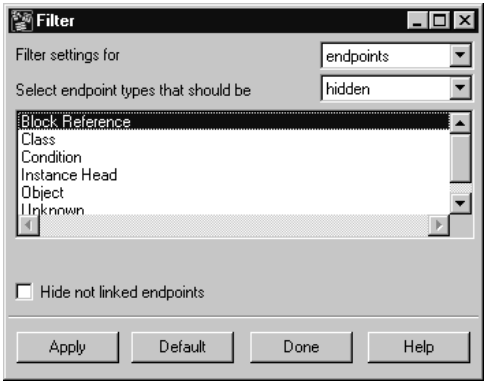


Figure 151: The Filter dialog

- *Filter settings for links/endpoints/documents*

  By using this option menu, filtering can be done on endpoint types, link names, and document file names. All filters are active simultaneously, i.e. for the filter to take effect, all three filter conditions must be true.

  The multiple-selection list in the dialog contains all link names, endpoint types or file names in the loaded link file, depending on the current setting of the option menu. By default, nothing is selected.

- Select <type of filter> that should be hidden/shown

  Depending on this option menu choice, the selected links/endpoints/documents will either be hidden or shown.

- *Hide not linked endpoints*

  If this option is set, all endpoints which are not linked will be hidden. This option is dimmed if filtering is not done on endpoint types.

- Pressing *Default* will set all the lists to their default values, but the filter is not applied until *Apply* is pressed.

Set Scale

Issues a dialog where the scale may be set.

Tools Menu

The *Tools* menu contains the following menu choices:

- *Show Organizer*
  (see "Show Organizer" on page 15 in chapter 1, *User Interface and Basic Operations*)

- *Search*
- *Search Again*
- *Consistency Check*
- *Check Endpoints*
- *Show in Editor*.

Search

This menu choice searches for a text string in endpoints or links.

The searching is based on ASCII character matching. All texts related to the endpoints and links are searched, i.e. endpoint types and names, link names and comments.

The search will start from the selected endpoint, or from the first endpoint if nothing is selected.

The following dialog appears:

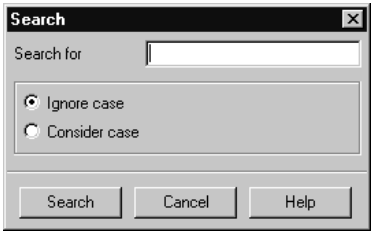


Figure 152: The Search dialog

- *Search for*

  The text string to search for. If a search already has been done, the previous search text is used by default.

- *Ignore Case/Consider case*

  Depending on the radio button setting, the search will either be case sensitive or not.

Search Again

This menu choice searches again for the same text string as the last performed search.

The menu choice is dimmed if a search has not yet been done.

Consistency Check

This menu choice is used for checking the consistency between a group of documents (the *FROM* group) and another group of documents (the *TO* group).

In Entity view, there are two types of consistency checks to choose from. The following dialog appears:

Figure 153: The Link Check/Entity Match dialog

- *Link check*

  Check that all endpoints/entities in the *FROM* group are linked with at least one endpoint/entity in the *TO* group.

- *Entity match*

  Check that all entities (not endpoints) in the *FROM* group has matching entities (not endpoints) in the *TO* group.

The above dialog is not opened in Endpoint view, in which case a link check always is performed.

First, the documents in the *FROM* group must be selected. The following dialog appears:

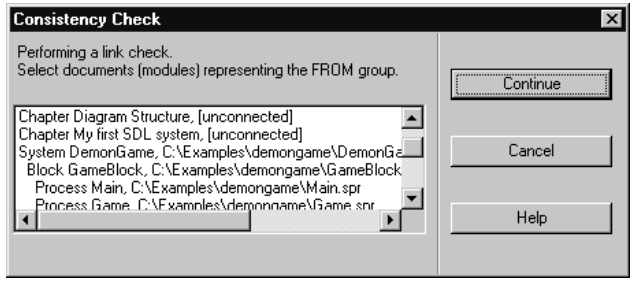


Figure 154: Selecting the FROM group

In the list of Organizer documents and modules, one or several documents must be selected. Selecting/deselecting a module will select/deselect all the documents in that module. Selecting/deselecting an SDL system will select/deselect all documents in that system. Individual documents in the module/system can then be selected/deselected without affecting the other documents.

When all *FROM* documents are selected, the *Continue* button is used to close the dialog and continue to the next dialog.

Then, the documents in the *TO* group must be selected. The following dialog appears:

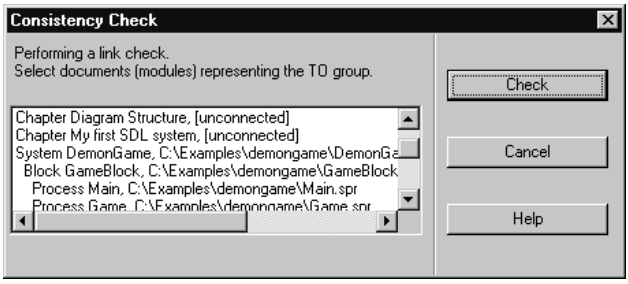


Figure 155: Selecting the TO group

The list of Organizer documents and modules works in the same way as when selecting the *FROM* documents. When all *TO* documents are selected, the *Check* button is used to close the dialog and start the consistency check operation.

The result of the consistency check is presented as a special view in the drawing area. This *Consistency view* only shows endpoints/entities from the two defined groups of documents. Endpoints/entities from the *FROM* group are shown in the normal way, and endpoints/entities from the *TO* group are shown as Dashed symbols.

The Consistency view is only showing links going from the *FROM* group of entities/endpoints to the *TO* group. After a link check, the links shown are the “real” links, as defined by the user. After an entity match, the links shown are only temporary links created by the Link Manager to indicate matching entities. The link name for such links are “Matching” and the link comment is “(Temporary link)”.

By selecting Endpoint or Entity view in *Options > Window*, the view will return to the selected normal view. By pressing the quick button *Show Endpoints or Entities*, the view will return to the previously used view.

Check Endpoints

This menu choice checks if there are endpoints in the Organizer or in the editors that do not exist in the Link Manager, or if there are invalid endpoints in the Link Manager. This menu choice could be used to remove any inconsistencies between the document endpoints in the system and the information in the Link Manager.

The information in the saved files is used for the checking, so if there are unsaved changes in an editor, these will not be taken into account in the checking. If there are any documents in the Organizer which have not been saved, the following dialog appears:

First, the Organizer and all documents in the system are checked. The following dialog appears:

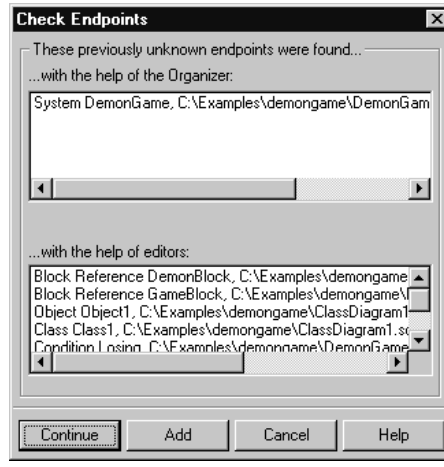


Figure 156: The first Check Endpoints dialog

In the upper multiple selection list, all endpoints that were found in the Organizer, but do not exist in the Link Manager, are listed. In the lower multiple selection list, all endpoints that were found in the documents belonging to the system, but do not exist in the Link Manager, are listed. It is possible to select one or more of the endpoints in the lists, with the purpose of adding them to the Link Manager.

- *Continue*

  Closes the dialog and continues to the next dialog. Any selected endpoints are **not** added to the Link Manager.

- *Add*

  Adds the selected endpoints to the Link Manager. The dialog is not closed until the *Continue* button is pressed.

Then, the endpoints in the Link Manager are checked. If any non-existing endpoints are found, they will be marked as Invalid. The following dialog appears:

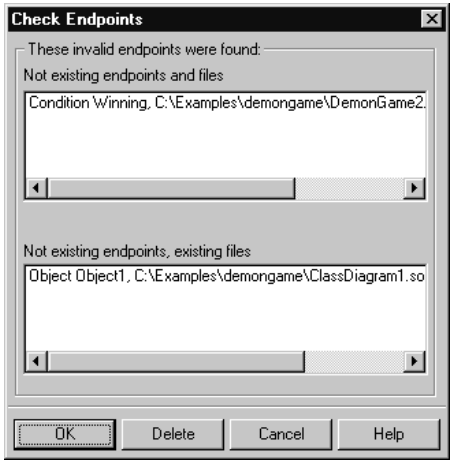


Figure 157: The second Check Endpoints dialog

In the upper multiple selection list, all Link Manager endpoints which reside in files that have been deleted are listed. In the lower multiple selection list, all Link Manager endpoints that no longer are present in the files they are supposed to be in are listed. It is possible to select one or more of the endpoints in the lists, with the purpose of deleting them from the Link Manager.

- *OK*

  Closes the dialog and returns to the main window. Any selected endpoints are **not** deleted from the Link Manager.

- *Delete*

  Deletes the selected endpoints from the Link Manager. The dialog is not closed until the *OK* button is pressed.

Show in Editor

This menu choice will show the symbol which corresponds to the selected endpoint in an editor.

In Entity view, each entity can represent more than one endpoint. In that case, the menu choice will be replaced by *Show 1 in Editor*, *Show 2 in Editor*, etc.

If there are more than nine endpoints, the menu choice *Show in Editor* will invoke the following dialog, where it is possible to select the appropriate endpoint:

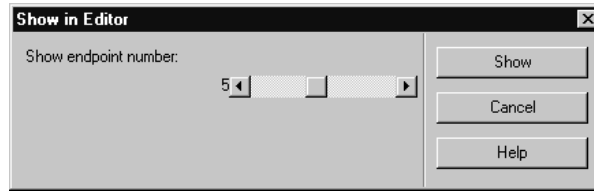


Figure 158: The Show in Editor dialog

Popup Menus

There are two popup menus available in the Link Manager:

On Endpoints

Highlight Endpoint	"Highlight Endpoint" on page 467
Replace Endpoint	"Replace Endpoint" on page 467
Create Link	"Create Link" on page 468
Link Details	"Link Details" on page 469
Clear Link	"Clear Link" on page 469
Expand	"Expand" on page 470
Expand Substructure	"Expand Substructure" on page 470
Collapse	"Collapse" on page 470
Show in Editor	"Show in Editor" on page 480

On the Background

Consistency Check	"Consistency Check" on page 476
Check Endpoints	"Check Endpoints" on page 478
Expand Substructure	"Expand Substructure" on page 470
Collapse	"Collapse" on page 470
Options > Window	"Options > Window" on page 471
Options > Link	"Options > Link" on page 472
Options > Endpoint	"Options > Endpoint" on page 473
Filter	"Filter" on page 474
Search	"Search" on page 475
Search Again	"Search Again" on page 476
Show Organizer	"Show Organizer" on page 15 in chapter 1, *User Interface and Basic Operations*

Keyboard Accelerators

In addition to the standard keyboard accelerators, described in "Keyboard Accelerators" on page 35 in chapter 1, *User Interface and Basic Operations*, the following accelerators can be used in the Link Manager:

Accelerator	Reference to corresponding command or quick button
`Ctrl+E`	"Show in Editor" on page 480
`Ctrl+1`	"Show Organizer" on page 15 in chapter 1, *User Interface and Basic Operations*
`Del`	"Clear Link" on page 469
`Arrow up`	Select the endpoint one step up (move the selection)
`Shift+arrow up`	"Move Up" on page 484
`Arrow down`	Select the endpoint one step down (move the selection)
`Shift+arrow down`	"Move Down" on page 484

Quick Buttons

Except for some of the general quick buttons (see "General Quick-Buttons" on page 24 in chapter 1, *User Interface and Basic Operations*) the following quick buttons are included the Link Manager.

Show Endpoints or Entities
Switches between the Endpoint and the Entity views; see "Options > Window" on page 471.

Consistency Check
Performs a consistency check operation; see "Consistency Check" on page 476.

Highlight Endpoint
Highlights the selected endpoint; see "Highlight Endpoint" on page 467.

Create Link
Creates a link between the highlighted and the selected endpoint; see "Create Link" on page 468.

Switch Link Direction
Changes the order of the links going to or from the (root) endpoint; see "Options > Link" on page 472.

Show or Hide Unused Endpoints
Toggles between showing and hiding endpoints that are not linked; see "Filter" on page 474.

Move Down
Moves the selected (root) endpoint one step down in the Link Manager view.

Move Up
Moves the selected (root) endpoint one step up in the Link Manager view.

The Link File

The Link Manager maintains a link file that contains a list of all endpoints and links in the system. The link file is a line-oriented, human-readable text file, with the default file name extension `.sli`.

The file has the following format:

```
<link file> ::= <endpoints> <links> $

<endpoints> ::= [ENDPOINTS] <endpoint>*
<endpoint> ::= <endpoint id> <endpoint format>
<endpoint id> ::= <integer>
```

A unique integer identifying the endpoint in the link file.

```
<endpoint format> ::= '(' <format> , <file name>
                      (, <anchor> | $) ')' <name>
                      <type>
<format> ::= SDL | OM | MSC | TEXT | WORD

<file name> ::= <string>
```

Name of the document file.

```
<anchor> ::= <string>
```

A unique string identifying the endpoint in the document. If the endpoint refers to the whole file, i.e. the endpoint is created in the Organizer, the anchor is set to an empty string ("").

```
<name> ::= <quoted string>
```

The name of the endpoint, in quotes.

```
<type> ::= <integer>
```

Type of document.

```
<links> ::= [LINKS] <link>*
<link> ::= <endpoint id> <endpoint id> <link name>
           <link comment>
<link name> ::= <quoted string>
```

The name of the link, in quotes.

```
<link comment> ::= <quoted string>
```

The link comment, in quotes.

Chapter 9 Implinks and Endpoints

Chapter

10 *The PostMaster*

This chapter is a reference to the internal communication mechanism between tools in SDL Suite and TTCN Suite, the PostMaster. The functionality of the PostMaster makes it possible to integrate applications by using a well-defined means of communication.

Introduction to the PostMaster

Caution!

The PostMaster was originally designed and implemented for integrating tools in the SDL Suite environment. Currently it integrates tools in the SDL Suite and the TTCN Suite environment. Our experience is that the PostMaster is also suitable for integrating external tools and applications with SDL Suite or TTCN Suite applications; one application area is for instance quick prototyping.

However, IBM Rational does not support using the PostMaster as a communication mechanism between real-time applications, in a run-time environment.

The PostMaster is the mechanism used for communication between the different tools in SDL Suite and TTCN Suite. A C program generated by the SDL Suite or the TTCN Suite tools can also take advantage of this communication mechanism. It can communicate with any application connected to the PostMaster that send messages according to a defined format. This makes it possible for an SDL simulator to communicate with, for instance, a user interface process for the Simulator.

The PostMaster also provides the basic means for an open public interface concept, see chapter 11, *The Public Interface*.

The PostMaster provides the following functionality:

- Starting an application and connecting to it
- Letting an application connect itself
- Letting an application send messages to a given recipient
- Making a “broadcast” of a message.

The PostMaster is a message passing service based on a selective broadcasting mechanism. It will distribute a copy of each message it receives to the tools subscribing to that type of message. By this, the PostMaster provides an integration mechanism between tools without the hard coupling between them that follows from conventional two part communication mechanisms.

Figure 159 illustrates some of the PostMaster concepts. The PostMaster maintains a list of which messages each tool subscribes to. Each tool has a PostMaster part for sending and receiving messages. In the figure, tool F broadcasts a message, i.e. the message is sent to the PostMaster. The

subscription lists of tool A and C (but not the lists of B and F) contains the message type. Accordingly the PostMaster broadcasts the message to tool A and C.

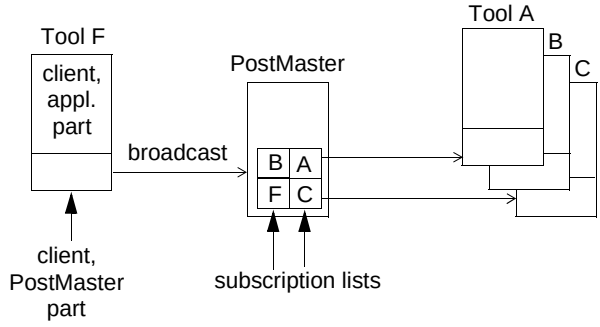


Figure 159: Example of a PostMaster broadcast

The PostMaster configuration is a file that informs the PostMaster about what tools and messages exist in the current context, i.e. it contains the message subscription lists. To include new tools or add new messages, the configuration must be edited.

For detailed information on the configuration, see "The PostMaster Configuration" on page 490.

PostMaster Reference

This section describes the external interface to the PostMaster, including messages to send and their format, contents of the configuration, and functions to call.

> **Note:**
>
> In order to promote a high throughput, it is strongly recommended that the PostMaster messages are consumed as soon as they are available.

PostMaster Messages

The PostMaster configuration defines a large number of messages that can be utilized in an SDL Suite or TTCN Suite system. A broad range of these messages are public, that is, they can be used externally. A description of these are found in *chapter 11, The Public Interface*.

All messages contain information about the identity of the sender, the time it was sent, the message type, and the size of an optional data part.

The optional data part can be seen as parameters to the message and is not interpreted by the PostMaster.

> **Note:**
>
> It is the responsibility of the tools using the PostMaster to define the format of the data part and to interpret it correctly.

The PostMaster Configuration

The PostMaster can be configured either statically at start up or dynamically during runtime. When a dynamic configuration is performed, the services Add Tool or Add Tool Subscription is used. The PostMaster is configured statically using the PostMaster configuration file(s)

The PostMaster configuration file(s) informs the PostMaster about what tools and messages exist in the current context. These files are often referred to as simply the *configuration*.

At start-up the PostMaster reads an environment variable `POSTPATH` which is a list of directories separated by colons **(on UNIX)** or semicolons **(in Windows)**. In these directories, the PostMaster searches for configuration files.

Such files should be named `post.cfd` and are read by the PostMaster whenever it is invoked, for instance when the SDL Suite or the TTCN Suite is started.

It is not possible to use C preprocessor statements or symbols in the configuration file.

Caution!

The file `post.cfd` residing in `$telelogic/sdt/bin` and `$telelogic/itex/bin` **(on UNIX)**, or in the installation directory **(in Windows)**, defines the existing tools in the environment and must not be edited; otherwise unpredictable behavior may occur.

It is possible to have local configurations which extends the standard PostMaster configuration set up by SDL Suite and TTCN Suite by simply defining the `POSTPATH` variable with a directory holding the extended configuration.

File Contents and Syntax

The configuration is a list of tool identities. Each tool identity is bound to an executable, a subscription list of messages and optionally a limit of instances of that tool. Each tool in the file is described in the following way:

```
Tool <tool number>:<executable>:<instance limit>
<message number>;
<message number>;
...
```

Description

- `<tool number>`

  is an integer defined by `SET_` symbols in the file `sdt.h`, and by `IET_` symbols in the file `itex.h`. The number 27000 (`SET_SDLENV`) is used for tools acting as an SDL environment.

- `<executable>`

  is the name of the executable file associated with the tool. It is used by the PostMaster after receiving an `SESTART` message when a tool is to be started.

  A file path may prefix the filename of the executable in order to explicitly tell where to find the tool. In this case the complete file description (path + filename) should be surrounded by a pair of quotes (i.e. `"/home/sdt/demo/demotool"` **(on UNIX)** or `"c:\sdt\demo\demotool"` **(in Windows)**).

- `<instance limit>`

  is the maximum number of concurrent running instances of the tool. It may be wildcarded with a '*' to a default value (`SPMAXNOOFINSTANCES`).

- `<message number>`

  is an integer defined by the symbols starting with `SE` in `sdt.h`. It is normally specified as `<tool number> + <nr>`. Symbols which are used in the TTCN Suite have the prefix `IE` and are defined in `itex.h`.

Adding Tools and Messages

To include new tools or add new messages, the configuration file must be edited. However, the tools and subscription lists existing in the original configuration must not be changed.

To **include a new tool**, follow the steps below. Note that it is not required to include a new tool in the configuration if it only serves as an SDL/TTCN environment that does not need to be started from other tools by using the Start service.

1. Select a tool number not conflicting with the ones already existing in the configuration, preferably a number greater than 100,000. The number chosen must be equally divisible by 1,000.
2. Make a new tool description in the configuration, using the syntax described above.
3. Define a new `SET_` (`IET_`) symbol in the file `sdt.h` (`itex.h`) so that the tool number can be easily accessed in the code.

Example 27 ----------

```
#define SET_MYTOOL   110000
```

To **add a new message**, follow these steps:

1. Decide upon a tool that the new message “belongs to” or “is defined by,” which should not be one of the pre-defined tools in the original configuration.
2. Define a new `SE` (`IE`) symbol in the file `sdt.h` (`itex.h`) as the tool symbol plus an ordinal number, not conflicting with any other message symbol.

Example 28 ----------

```
#define SEMYMESSAGE   SP_MESSAGE(SET_MYTOOL+1)
```

---

3. Use the numeric value of the message symbol when adding the new message to a subscription list in the configuration.

Example 29 ----------

```
110000+1;
```

Environment Variables

The PostMaster recognizes a number of environment variables setting the context in which the PostMaster is to operate. They are read when the PostMaster starts.

PostMaster Environment Variables

The following environment variables are recognized by the PostMaster:

- `POSTPORT`

  UNIX only: Should be set to a valid TCP port number. Causes the PostMaster to try to listen to this port number for possible connections. Makes it possible to connect applications running on other hosts than the PostMaster is running on. Is used in combination with the client side environment variables `POSTHOST` and `POSTPORT`.

- `POSTPATH`

A list of directories separated by colons **(on UNIX)** or semicolons **(in Windows)**, where to search for configuration files named `post.cfd`.

- `POSTDEBUG`

  If set, the PostMaster will log active tools and sent messages as an MSC log. The environment variable takes two optional parameters. The first tells the filename to put the log on. If the parameter is set to `'-e'` the log is put on `stdout`. The second parameter tells the log level. Normally only public messages are logged. But if set to a value > 2 all messages will be logged.

  If no parameters are submitted, the log will be stored on the file `post.mpr`

  See also "Start MSC Log" on page 596

- `STARTTIMEOUT`

  When the PostMaster executes its start service it assumes that the started client will connect, via SPInit, within a certain amount of time. This timeout (in seconds) sets the limit when the PostMaster considers the start having failed. Default is 60 seconds.

  If the SDL Suite and TTCN Suite tools environment is running on slow computer or on a network which is heavily loaded, start-up of a tool might fail due to this timeout. In such case, this environment variable should be increased to an appropriate value.

PostMaster Application Library Environment Variables

The following environment variables are recognized by the PostMaster application library. These environment variables are read in the SPInit function.

- `POSTHOST`

  UNIX only: Should be set to a hostname, on which a PostMaster runs. This PostMaster should be started with the environment variable `POSTPORT` set.

- `POSTPORT`

  UNIX only: Should be set to a valid TCP port which is allocated to a PostMaster. Corresponds to `POSTPORT` set for the PostMaster.

- `POSTPID`

  This environment variable is only of interest if multiple PostMasters are simultaneously active. If it is desired to make a connection to one particular PostMaster, this environment variable should be set to the process id corresponding to that PostMaster.

  If only one PostMaster is active, the application library automatically finds the PostMaster.

  If more than one PostMaster is running and this environment variable is not set, a connection is made to the PostMaster instance having the highest process id value.

Functional Interface

Communication with the PostMaster is based on a small set of fundamental functions:

SPInit	initialization
SPExit	termination
SPBroadcast	output message, broadcast
SPSendToTool	output message, sent to a certain tool
SPSendToPid	output message, sent to a certain PId
SPRead	input message
SPFree	frees memory allocated by SPRead or SPFindActivePostMasters
SPConvert	translate a symbolic message to an id
SPErrorString	Error string conversion
SPRegisterPMCallback	Message notification **(Windows only)**
SPQuoteString	Quoting strings
SPUnquoteString	Unquoting strings
SPFindActivePostMasters	find all PostMasters

Every function returns a value denoting success or failure of the associated operation.

Calling Conventions

The PostMaster functions should be called in the following ways from the external tool:

- The first thing to do is to initiate the connection with the PostMaster itself by calling SPInit. This is preferably done early in the main program. This initiation is private to the client and the PostMaster.
- When having successfully called SPInit, the tool should listen to the PostMaster message port and read the incoming messages by SPRead. The tool should then act upon the messages it subscribes on.
- If so desired, the tool should send messages by calling SPSendToTool, SPSendToPid or SPBroadcast.
- When the tool decides to terminate, it should broadcast the standard message SESTOPNOTIFY by calling SPBroadcast. Then it should disconnect from the PostMaster by calling SPExit.
- Every call to a PostMaster function should check the return value and act properly if an error is detected.

Variables

The following variables are defined.

```
extern INT16 sperrno;

extern INT16 spPortNO; (on UNIX)
extern int spPortNO; (in Windows)
```

sperrno contains an error code when a call to a PostMaster function failed. The error codes are defined in sdt.h and should be self-explanatory.

On UNIX, spPortNO contains a descriptor to the port where incoming messages from the PostMaster are found. **In Windows**, spPortNO contains a process identifier associated with the current process. The state of the variable prior to the call of SPInit is undefined.

These variables are found in the file post.h.

Caution!

Windows only: When linking with the PostMaster's dynamically linked libraries (`post.lib` and `post.dll`), the environment variable `USING_DLL` must be defined before including `post.h`. Example:

```
#define USING_DLL
#include "post.h"
#undef USING_DLL
```

Error Codes

When a function returns `SPERROR` which indicates an error or an not expected result of the function. In this case `sperrno` is set to one of the following values.

Error Code	Explanation
`SPNOSESSION`	The function `SPInit` has not been called successfully
`SPALREADYCONNECTED`	When calling `SPInit` more than once without disconnecting
`SPNOPOSTMASTER`	No PostMaster could be found when trying to connect
`SPNOCHANNEL`	The contact with the PostMaster is lost.
`SPNOMESSAGE`	No message available when trying to read by polling or after a timeout.
`SPTIMEDOUT`	The connection to the PostMaster timed out.
`SPNOSUCHPID`	Sending to a PId with a non positive value (<=0)
`SPNOMEMORY`	Cannot allocate anymore dynamic memory (rare)
`SPTOOLNOTFOUND`	When sending to a tool, this tool is not found in PostMaster configuration list

Error Code	Explanation
SPINVALIDMESSAGE	Sending a message with a NULL parameter but specifying the length greater than 0.
SPBADMESSAGE	A not supported message was to be sent.
SPMANYPOSTMASTERS	Many PostMasters running. Could not decide which one to connect to. **(Windows only)**
SPOLDPOSTMASTER	Old PostMaster running. **(Windows only)**

Functional Description

The available PostMaster functions are described on the following pages. The descriptions use the following format.

First, the function declaration is shown, including data types of parameters, followed by a short explanation of what the function does.

After that, in- and out parameters are described, together with possible return values and error codes.

SPInit

```
int SPInit(int toolType,
           char * argv0,
           *SPMList list);
```

`SPInit` initiates a session and establishes a connection with the PostMaster. See "Multiple PostMaster Instances" on page 518 for information on how to connect to a specific instance of the PostMaster.

Inparameters

- `toolType`

  The tool number identifies the tool and should be a value available in the PostMasters subscription list. For tools acting as an SDL/TTCN environment, it should be set to `SET_SDLENV`. See "Run-Time Considerations" on page 515 for more information.

- `argv0`

  The name of the executable (specified with its path). Normally argv[0] as passed to the program could be used.

- `list`

  Defines a list of predefined messages allowed to send. It also provides a set of mappings between textual strings and integer values for tools and messages. Normally the list provided in `sdt.h` (`itex.h`) is used. This list is later used by the function `SPConvert`, which translates between a textual string and the corresponding identifier.

Returns

- `>0`

  On UNIX, the PId of the calling process. Normally this PId corresponds to an UNIX PId. But if the PostMaster is started with the environment variable `POSTPORT` set, the PostMaster decides what PId value each client should get. In this case the numbering scheme gives the first started client 1, the next 2 and so on.

  In Windows, an identifier ("PId") which is used internally by the PostMaster in order to uniquely identify the calling process.

  This value could be used when sending messages and in comparisons with PIds contained in received messages.

- SPERROR

  SPInit failed, sperrno is set.

Errors

SPNOPOSTMASTER
SPTIMEDOUT
SPALREADYCONNECTED
SPNOMEMORY
SPMANYPOSTMASTERS
SPOLDPOSTMASTER

SPExit

```
int SPExit(void);
```

SPExit exits a session and disconnects the connection with the PostMaster. Subsequent calls to PostMaster functions will return the error code SPNOSESSION until a new SPInit is performed.

Returns

- SPOK

  Status OK.

- SPERROR

  SPExit failed, sperrno is set.

Errors

SPNOCHANNEL
SPNOSESSION

SPSendToTool

```
int SPSendToTool(int tool,
                 int event,
                 void * data,
                 int size);
```

`SPSendToTool` sends a message to the process of kind `tool`.

Inparameters

- `tool`

  Type of tool identifying the tool to send the message to. If such a tool is not running a service reply is sent by the PostMaster.

- `event`

  Type of message

- `data`

  Handle to an information block

- `size`

  Size of data.

Returns

- `SPOK`

  Status OK.

- `SPERROR`

  `SPSendToTool` failed, `sperrno` is set.

Errors

`SPNOCHANNEL`
`SPNOSESSION`
`SPNOMEMORY`
`SPBADMESSAGE`
`SPINVALIDMESSAGE`
`SPTOOLNOTFOUND`
`SPBADMESSAGE`
`SPNOSUCHPID`

SPSendToPid

```
int SPSendToPid(int pid,
                int event,
                void * data,
                int size);
```

`SPSendToPid` sends a message to the process which has process id `toPid`.

Inparameters

- `pid`

  PId of the message's receiver. If the specified PId does not exist, an `SEOPFAILED` message or a service reply is sent by the PostMaster.

- `event`

  Type of message.

- `data`

  Handle to an information block.

- `size`

  Size of data.

Returns

- `SPOK`

  Status OK.

- `SPERROR`

  `SPSendToPid` failed, `sperrno` is set.

Errors

`SPNOCHANNEL`
`SPNOSESSION`
`SPNOMEMORY`
`SPBADMESSAGE`
`SPINVALIDMESSAGE`
`SPTOOLNOTFOUND`
`SPNOSUCHPID`

SPBroadcast

```
int SPBroadcast(int event,
                void * data,
                int size);
```

`SPBroadcast` sends a message to all processes that subscribes on the message type.

Inparameters

- `event`

  Type of message.

- `data`

  Handle to an information block.

- `size`

  Size of data.

Returns

- `SPOK`

  Status OK.

- `SPERROR`

  `SPBroadcast` failed, `sperrno` is set.

Errors

SPNOCHANNEL
SPNOSESSION
SPNOMEMORY
SPBADMESSAGE
SPINVALIDMESSAGE

SPRead

```
int SPRead(int timeOut,
           int * pid,
           int * message,
           void ** data,
           int * len);
```

SPRead reads a message from the queue of unread messages that the PostMaster has sent. When the message is read, it is also consumed, i.e. removed from the queue. The function allocates the necessary amount of memory needed for the data component in the message. The application using this function is responsible for freeing the allocated memory with SPFree when it is no longer needed.

Inparameters

- timeOut

  Maximum amount of time (in milliseconds) that the function waits for a message. If a message has not arrived when timeOut expires, the function returns with return value SPERROR and sperrno is set to SPNOMESSAGE. If a message arrives, the function reads the message and returns immediately. If the desired behavior is to wait until a message arrives, which could mean forever, timeOut should be set to SPWAITFOREVER.

Outparameters

- pid

  PId of the tool sending the message.

- message

  Message identifier.

- data

  Pointer to data associated with the received message. SPFree should be used to free memory.

  Length of allocated data. If ASCII data is received, it is not assured that data is terminated by a ASCII NUL (‘\0’). The application should test if data[len-1] is ‘\0’.

Returns

- SPOK

Status OK.

- SPERROR

  SPRead failed, sperrno is set.

Errors

SPNOCHANNEL
SPNOMESSAGE
SPNOSESSION
SPNOMEMORY

SPFree

```
void SPFree (void * ptr)
```

SPFree should be used to free the memory allocated by SPRead. This is necessary when different compilers with different memory management are used.

Inparameters

- ptr

  A pointer to the memory block to be freed.

Returns

N/A.

SPErrorString

```
char * SPErrorString(int code);
```

Converts an error code into a textual string description of a tool or a message from the corresponding integer value.

Inparameters

- code

  Error code. Typically set by sperrno.

Returns

An descriptive error string corresponding to the error code.

SPConvert

```
int SPConvert(char * str);
```

Converts a textual description of a tool or a message to the corresponding integer value as provided by the parameter `list` in `SPInit`.

Inparameters

- `str`

  Textual description of the tool or message. The list provided in `SPInit` is used when searching for a mapping.

Returns

An integer value for the tool or message. If no mapping is found, `SPERROR` is returned.

SPRegisterPMCallback

```
typedef void (* SP_PM_MessageCallback) (void);
void SPRegisterPMCallback (SP_PM_MessageCallback
cb);
```

Windows only: Registers a callback function that gets called every time a new PostMaster message arrives. Registering this callback enables (32-bit) Windows applications to function correctly. Console applications need not use this function.

Inparameters

- `cb`

  The function to be called when a new PostMaster message is present. If `cb` is `NULL` the current callback is removed; otherwise the callback is replaced.

Returns

N/A.

SPQuoteString

```
int SPQuoteString(char *stringToQuote,
                  char *buffer,
                  int bufferLength,
                  int append)
```

`SPQuoteString` quotes a string. The following operations are performed:

- A quote is added to the beginning and end of the string.
- All quotes and backslashes in the string are escaped, i.e. a backslash character is added before them.

The quoted string can later be unquoted with a call to SPUnquoteString.

Inparameters

- `stringToQuote`

  Pointer to a null-terminated string. This is the string that should be quoted.

- `bufferLength`

  The size of the buffer (see the outparameter `buffer` below).

- `append`

  If the value of append is non-zero, `buffer` is supposed to already contain a null-terminated string. The result of the quoting operation will be appended to this string at the end.

  If several quoted strings are concatenated, they can be extracted and unquoted one at a time with calls to the SPUnquoteString function.

Outparameters

- `buffer`

  Pointer to a buffer where the resulting, quoted string will be returned. The buffer must be large enough to contain all the characters of the quoted string plus a trailing null character. The maximum buffer size needed can be quoted with the following formula:

  maximum buffer size = unquoted string size * 2 + 3

This includes space for escaping every character in the string, plus three bytes for quotes and null character. A larger buffer might of course be needed when appending (see `append` above).

The size of the buffer is given in `bufferLength`. If the quoted string does not fit in the buffer, the function will fail and return zero (further explained in the next section).

Returns

- `1`

  The call was successful.

- `0`

  The function call failed. The buffer was not large enough to contain the quoted representation of the string. The contents of the buffer are undefined.

SPUnquoteString

```
int SPUnquoteString(char *quotedString,
                    int inputLength,
                    char *buffer,
                    int bufferLength,
                    int position)
```

`SPUnquoteString` unquotes a string previously quoted with SPQuoteString, ignoring leading white-space characters. The following operations are performed:

- Any white-space characters in the beginning of the string are ignored. If the string is empty or contains only white-space characters, the function call fails and returns zero (see "Returns" on page 510).
- If the first character after white-spaces is a quote, the function assumes that this quote starts a string quoted with SPQuoteString. In this quoted string, backslashes escape the following character. The string is ended with a non-escaped quote.

  The returned string will have the escaping backslashes and the leading and closing quotes removed.
- If the first character after white-spaces is **not** a quote, the function will simply extract and return a substring up to but not including the next white-space character. If no further white-space characters are found, the rest of the string is returned. Backslashes and quotes have no special meaning when unquoting strings in this way.

Several concatenated quoted strings can be extracted with subsequent calls to this function by using the value returned in the `position` out parameter, see below.

Inparameters

- `quotedString`

  Pointer to a string containing the string previously quoted with SPQuoteString. The string should either be null-terminated or `inputLength` below should have a meaningful value.
- `inputLength`

  The maximum number of characters of `quotedString` that will be scanned. If `quotedString` is known to be null-terminated, a very large number should be supplied here.

- `bufferLength`

  The size of the buffer (see the outparameter `buffer` below).

Outparameters

- `buffer`

  A pointer to a buffer where the unquoted string will be returned. The buffer must be large enough to contain the resulting, unquoted string, including null character. A buffer one character larger than `inputLength` will always suffice. If the buffer is too small, the function call will fail and return zero (see "Returns" on page 510). The size of the buffer is given in `bufferLength`.

  `buffer` can also be null, in which case no unquoting will be performed. The value of `position` can still be interesting, though.

- `position`

  If this pointer is non-null it should point to an integer that will be filled in with the index of the character in the input string immediately following the extracted substring. This will be the character following a closing quote or a whitespace character or the terminating null character.

Returns

- `1`

  The function call was successful.

- `0`

  The function call failed. The supplied string might not be a valid quoted string according to the rules given above. The buffer might not be large enough to contain the result. The contents of the buffer and `position` are undefined.

SPFindActivePostMasters

```
int SPFindActivePostMaster (int *bufferPid,
                            char** bufferText,
                            int maxBufferlength);
```

Finds all PostMasters available for the application on the computer. Information retrieved is process id and a description in plain text.

Inparameters

- `maxBufferlength`

  The size of the arrays `bufferPid` and `bufferText`.

Outparamters

- `bufferPid`

  Pointer to an array where all process ids will be stored.

- `bufferText`

  Pointer to an array of strings telling when the PostMaster was started. `SPFindActivePostMaster` will allocate memory for all strings and the application must call SPFree in order to free the memory.

Returns

Number of PostMasters found.

Java Interface

The file `postmaster.java` contains a java class that encapsulates the postmaster interface for java programmers. The class contains a few fundamental methods.

`Init`	initialization
`SendToTool`	output message, sent to a certain tool
`SendToPid`	output message, sent to a certain Pid
`Broadcast`	output message, broadcast
`Read`	read a message
`Exit`	termination

Init

```
int Init(int toolType);
```

`SPInit` initiates a session and establishes a connection with the PostMaster. See "Multiple PostMaster Instances" on page 518 for information on how to connect to a specific instance of the PostMaster.

Inparameters

- `toolType`

  The tool number identifies the tool and should be a value available in the PostMasters subscription list.

Returns

See functional description for the C interface

SendToTool

```
int SendToTool(int toolType, int message,
               String data);
```

`SPSendToTool` sends a message to the process of kind `tool`.

Inparameters

- `toolType`

  Type of tool identifying the tool to send the message to. If such a tool is not running a service reply is sent by the PostMaster.

- `message`

  Type of message

- `data`

  Data to send

Returns

See functional description for the C interface

SendToPid

```
int SendToPid(int pId, int message, String data);
```

`SPSendToPid` sends a message to the process which has process id `toPid`.

Inparameters

- `pid`

  PId of the message's receiver. If the specified PId does not exist, an `SEOPFAILED` message or a service reply is sent by the PostMaster.

- `message`

  Type of message

- `data`

  Data to send

Returns

See functional description for the C interface

Broadcast

```
int Broadcast(int message, String data);
```

`SPBroadcast` sends a message to all processes that subscribes on the message type.

Inparameters

- `message`

  Type of message

- `data`

  Data to send

Returns

See functional description for the C interface

Read

```
int Read(int timeOut);
```

`SPRead` reads a message from the queue of unread messages that the PostMaster has sent. When the message is read, it is also consumed, i.e. removed from the queue. The data read by the last `Read` is copied to the `Sender`, `Message` and `Data` members of the class `postmaster`.

Inparameters

- `timeOut`

Maximum amount of time (in milliseconds) that the function waits for a message. If a message has not arrived when `timeOut` expires, the function returns with return value `SPERROR` and `sperrno` is set to `SPNOMESSAGE`. If a message arrives, the function reads the message and returns immediately. If the desired behavior is to wait until a message arrives, which could mean forever, `timeOut` should be set to `SPWAITFOREVER`.

Returns

See functional description for the C interface

Exit

```
int Exit();
```

`SPExit` exits a session and disconnects the connection with the PostMaster. Subsequent calls to PostMaster functions will return the error code `SPNOSESSION` until a new `SPInit` is performed.

Returns

See functional description for the C interface

Run-Time Considerations

Starting Up the PostMaster (in Windows)

When SDL Suite or TTCN Suite is started **in Windows**, an instance of the PostMaster is automatically started. No additional commands are needed for the PostMaster. There might however be situations when it is necessary to start the PostMaster stand-alone. This is described in the following sections. The examples only handle how the SDL Suite is started from the “DOS” command prompt, but in many cases a shortcut with the analogous parameters can be created.

Start-Up

There are several possible ways to start the PostMaster itself and the tools that wish to communicate via the PostMaster. In principle, there are two main alternatives:

- Starting the tools when the PostMaster is running
- Starting them without having the PostMaster running.

We will exemplify the start-up methods by using the DemonGame simulator and a User Interface to the DemonGame simulator.

> **Note:**
>
> The PostMaster **must be started first**, but the communicating tools may be started in any order. An SDL simulator must also execute the commands Start-SDL-Env and Go **before communication** with another tool can start.

Starting When the PostMaster Is Present

When the SDL Suite is started from the “DOS” or UNIX prompt, an instance of the PostMaster is automatically started. No additional commands are needed for the PostMaster in this case.

The DemonGame simulator is preferably started from the SDL Suite, thus giving access to all simulation features. It can also be started directly from the “DOS” or UNIX prompt.

Starting from the “DOS” Prompt

Use the following command from the “DOS” prompt:

```
DemonGame.exe -post
```

Note:

In Windows, the simulator is a Windows GUI application and not a Console application. It is therefore not possible to run the simulator stand-alone, i.e. without `-post`.

Starting from the UNIX Prompt

Use the following command from the UNIX prompt:

```
DemonGame.sct -post
```

(Without the parameter, the simulator runs stand-alone, which is not desired in this case.) Starting the simulator this way restricts the possibilities of the simulation since there is no connection to the SDL Suite tools. For instance, graphical trace is disabled.

The UI is preferably started directly from the UNIX prompt. It can also be started from the DemonGame simulator with the command Start-ITEX-Com, but this **requires** that the executable is named `sdtenv`. This is the name specified in the configuration for tool number 27000.

Starting Without the PostMaster

The SDL Suite must be started directly from the “DOS” or UNIX prompt with the command:

```
$telelogic/bin/sdtpm <arg> &
```
(on UNIX)

```
<Installation Directory>\sdt <arg>
```
(in Windows)

(using one of the start-up arguments, see below).

The DemonGame simulator must also be started directly from the “DOS” or UNIX prompt, as described above.

Note:

On UNIX: If activating the PostMaster this way, the environment variable `POSTPATH` **must** be set to include the directory where the executables resides, typically `$sdtbin`.

The UI can be started in the same way as when the PostMaster is running (see above). It can also be started indirectly when starting the PostMaster by using the “DOS” or UNIX command:

```
$telelogic/bin/sdtpm -clients 27000 & (on UNIX)

<Installation Directory>\sdt -clients 27000 (in Windows)
```

This **requires** that the executable is named `sdtenv`, the name specified in the configuration for tool number 27000.

> **Note:**
>
> A tool communicating through the PostMaster can also be started from another tool by using the `SESTART` message. This requires that the tool to be started is specified in the configuration.

Start-Up Arguments

The PostMaster recognizes the following arguments at start-up. Normally they are not needed, but could be used for special purposes

- `-clients <toolid>` Used if the PostMaster should invoke a certain tool at start-up. `<toolid>` is set to the logical tool number of a tool to start as defined in `post.cfd`.
- `-noclients` Used if only the PostMaster is to be invoked without starting any clients. In this case the PostMaster enters an idle state where it waits for a client to connect

All other arguments are passed to the tool started.

SDT-2 Connections (UNIX only)

Applications or tools linked with an SDT 2.X PostMaster application library will not be compatible with SDT 3.X PostMaster. Trying to connect such an old application to an SDT 3.X PostMaster will result in an error message:

```
Postmaster cannot connect SDT2 tool:<tool> with pid:
<pid>
```

on standard output, if externally started, or as a service reply, if started via the start service, and the connection is aborted.

Version 3.4 PostMaster (Windows only)

If an old PostMaster (version 3.4 or older) is used the SPInit function will fail. The error code is SPOLDPOSTMASTER.

Applications or tools linked with an SDT 3.4 PostMaster application library will not be compatible with the SDT 3.6 PostMaster. Trying to connect such an old application to an SDT 3.6 PostMaster with the SPInit function will fail. The error code is SPNOPOSTMASTER.

Multiple PostMaster Instances

It is possible to have multiple instances of the PostMaster running, for instance when more than one SDL Suite or TTCN Suite has been started. In this case it should be made sure that the communicating tools are connected to the correct instance of the PostMaster.

On UNIX, the function `SPInit` by default looks for the PostMaster instance with the **highest process id** (PId) number and connects to it.

On Windows, the function `SPInit` by default fails if more than one PostMaster is found.

To connect to another PostMaster instance, you can set the environment variable `POSTPID` to the PId number of the desired PostMaster. If this variable is set, `SPInit` connects to the PostMaster instance with that PId number.

Configuration and Tool Search

The PostMaster configuration is read whenever a PostMaster instance is invoked. The environment variable `POSTPATH` defines a list of directories where the PostMaster looks for a configuration file named `post.cfd`.

The user can extend the normal configuration by defining the `POSTPATH` variable to a directory containing an extended configuration. When SDL Suite or TTCN Suite is started normally, the directories containing the binaries are put as the first directories in the `POSTPATH` variable.

The same search order is applied when the start service is used. The supplied tool number is matched against the name of an executable in the configuration, which is then searched for as above.

Chapter

11 *The Public Interface*

This chapter introduces the Public Interface and describes a message based interface by which external tools and applications could be integrated with SDL Suite and TTCN Suite tools.

Three conceptually different facilities are provided:

- **By controlling the behavior of the tools and by modifying the data that are handled by the tools in SDL Suite and in TTCN Suite.**
- **By listening to important events of status change in SDL Suite and TTCN Suite.**
- **By integrating an external application with SDL simulators.**

The Public Interface uses the PostMaster as transport. This chapter assumes the reader to be familiar with the PostMaster; see chapter 10, *The PostMaster*.

In *chapter 12, Using the Public Interface*, you may find examples of how to use the Public Interface.

General Concepts

Introduction

Significant steps have been made to increase the openness of SDL Suite and TTCN Suite tools by the introduction of the *Public Interface*. Openness is provided both in terms of diverse ways of controlling the tools and the way data handled by the tools is made visible.

Application Areas

Two different kinds of usage of the Public Interface could be foreseen:

- As an alternate or complementary means to access functionality provided by the tools, compared to the normal graphical interface. Such usage could be to integrate simulators generated by the tools with dedicated user interfaces, or to use the tools batch facilities using make or print in a large “build” environment.
- Integrating SDL Suite and TTCN Suite tools with other tools forming a larger environment.

The Public Interface

The public interface has been made possible due to the modularity of the tools and thanks to the well defined interfaces by which the tools have been built.

The public interface is made available using two different mechanisms.

- Via a programming interface, providing message based services and notifications. The rest of this section provides a description of these services and notifications and their usage.
- Via applications executed via the OS command line interface. By using this interface:
  - Important services could be executed running in a *batch* mode.
  - A Service Encapsulator could be run, which allows message based services to be issued. This tool is also made available as a an example application. For more information, see “The Service Encapsulator” on page 537.

The PostMaster

The PostMaster forms an integral mechanism to integrate tools in the SDL Suite and TTCN Suite tools environment (a reference to the PostMaster is provided in chapter 10, *The PostMaster*).

It is also used as a low level mechanism to physically integrate SDL Suite and TTCN Suite tools with external tools. The PostMaster provides three basic facilities:

- It maintains a list on known tools. Such tools could be connected to the PostMaster and thereby taking part of message sending
- It provides means to broadcast messages to a list of subscribers
- It contains functionality to send a message to a certain recipient.

These facilities are used to implement the concepts provided by the SDL Suite and the TTCN Suite tools services. These are built up of *Services* and *Notifications*. A special case of service is *Communication with SDL Simulators*.

It is important to note that the interface provided could be extended for client-client usage outside SDL Suite and TTCN Suite as long as the conventions described in this document are followed.

Services

Services provides access to functionality within the SDL Suite and the TTCN Suite. Such functions could be used by an external tool to exercise control, to integrate SDL Suite or TTCN Suite with external tools or have SDL Suite and TTCN Suite take part in a larger environment.

Services adopts the client-server concept where a *request* is sent by a client to a server, which returns with a *reply*. Service request and service reply take both advantage of message sending from one tool to another.

Notifications

These are broadcast messages which are spontaneously emitted when a significant event takes place in SDL Suite or TTCN Suite. Often these notifications are caused by a service being successfully processed.

Communication with SDL Simulators

When external applications are to communicate with SDL Simulators, the message `SESDLSIGNAL` is used. It is asynchronously sent/broadcast-

er into the system. All simulators subscribes on this message. As a parameter to this message is the receiver (in the context of SDL) provided.

Overview of Available Services

The following services are provided for external usage. A service is normally supported by a specific tool or a few of the tools. Some of the services correspond to a menu choice or an operation performed by a user using the tools's graphical interface.

In the following services, the Configuration Services and the System File Services are applicable to SDL Suite and TTCN Suite, and the TTCN Suite Services are only applicable to the TTCN Suite. All other services are only applicable to SDL Suite.

Configuration Services

Service	Servers	Graphical correspondence
Start Tool	PostMaster	Start a new tool
Stop Tool	All tools	*Exit* menu choice
Get Tool Type	PostMaster	N/A
Get Tool Pid	PostMaster	N/A
Add Tool	PostMaster	N/A
Add Tool Subscription	PostMaster	N/A

System File Services

Service	Servers	Graphical correspondence
List System Files	Organizer	N/A
New System	Organizer	*New* quick button
Open System	Organizer	*Open* quick button
Save System	Organizer	*Save* quick button
Add Existing	Organizer	*Add Existing* menu choice

Link File Services

Service	Servers	Graphical correspondence
Add Local Link File	Organizer	N/A

Service	Servers	Graphical correspondence
Merge Local Link File	Organizer	N/A

TTCN Suite Services

Service	Servers	Graphical correspondence
Convert to GR	ITEX	N/A
Opened Documents	ITEX	N/A
Fetch Buffer Identifier Given the Database	ITEX	N/A
Fetch Buffer Identifier Given MP File Path	ITEX	N/A
Convert to MP	ITEX	N/A
Convert Selection to MP	ITEX	N/A
Merge Document	ITEX	N/A
Analyze Document	ITEX	N/A
Close Document	ITEX	N/A
Save Document	ITEX	N/A
Selector	ITEX	N/A
SelectAll	ITEX	N/A
DeselectAll	ITEX	N/A
IsSelected	ITEX	N/A
Get Modify Time	ITEX	N/A
Get Path	ITEX	N/A
Get MP Path	ITEX	N/A
Find Table	ITEX	N/A
Close Table	ITEX	N/A
Get Table State	ITEX	N/A
Get Row Number	ITEX	N/A

Service	Servers	Graphical correspondence
Select Row	ITEX	N/A
Clear Selection	ITEX	N/A
Rows Selected	ITEX	N/A

Menu Manipulation Services

Service	Servers	Graphical correspondence
Add Menu	Organizer SDLE MSCE OME TE Coverage Viewer Index Viewer Type Viewer Tree Viewer File Viewer Preference Manager SimUI/ExpUI	N/A
Delete Menu	As for Add Menu above.	N/A
Clear Menu	As above.	N/A
Add Item to Menu	As above.	N/A
Add Item to Menu – Organizer	Organizer	N/A
Add Item to Menu – Text Editor	TE	N/A
Add Item to Menu – Graphical Editors	SDLE MSCE OME	N/A

Logging Services

Service	Servers	Graphical correspondence
Start MSC Log	PostMaster	N/A
Stop MSC Log	PostMaster	N/A

SDT Reference Services

Service	Servers	Graphical correspon-dence
Show Source	Organizer	*Show source* menu choice
Obtain GR Reference	SDLE MSCE OME	N/A

Editor – Diagram Services

Service	Servers	Graphical correspondence
Load Diagram	SDLE MSCE OME TE	*Open* menu choice
Unload Diagram	SDLE MSCE OME TE	*Close* menu choice
Show Diagram	SDLE MSCE OME TE	*Diagrams* menu
Save Diagram	SDLE MSCE OME TE	*Save* menu choice
Create SDL Diagram	SDLE	*New* menu choice

Service	Servers	Graphical correspondence
Create MSC Diagram	MSCE	*New* menu choice
Create OM Diagram	OME	*New* menu choice
Create Text Diagram	TE	*New* menu choice

Editor – Object Services

Service	Servers	Graphical correspondence
Select Object	SDLE MSCE OME	*Selecting* an object
Show Object	SDLE MSCE OME	*Selecting* an object
Insert SDL Object	SDLE	*Select* object in symbol menu
Insert MSC Object	MSCE	*Select* object in symbol menu
Remove Object	SDLE MSCE OME	*Clear* menu choice
Get Object Text	SDLE MSCE OME	N/A

Editor – Object Attribute Services

Service	Servers	Graphical correspondence
Display Key	SDLE MSCE OME	N/A
List Key	SDLE MSCE OME	N/A

Service	Servers	Graphical correspondence
Create Attribute	SDLE MSCE OME	N/A
Update Attribute	SDLE MSCE OME	N/A
Read Attribute	SDLE MSCE OME	N/A
Delete Attribute	SDLE MSCE OME	N/A

Information Server Services

Service	Servers	Graphical correspondence
Load Definition File	Information Server	N/A

SDL Editor Services

Service	Servers	Graphical correspondence
GRPR	SDLE	N/A
Tidy Up	SDLE	*Tidy Up* menu choice

SC Editor Services

Service	Servers	Graphical correspondence
Get Diagram Info	OME	N/A

MSC Editor Services

Service	Servers	Graphical correspondence
MSC GRPR	MSCE	*Generate MSC PR* menu choice

HMSC Editor Services

Service	Servers	Graphical correspondence
HMSC GRPR	MSCE	*Generate MSC PR* menu choice

CIF Services

Service	Servers	Graphical correspondence
Create SDL Diagram	SDLE	N/A
Create SDL Page	SDLE	N/A
Insert SDL Object	SDLE	N/A
Create OM Diagram	OME	N/A
Create OM Page	OME	N/A
Insert OM Object	OME	N/A

Text Editor Services

Service	Servers	Graphical correspondence
Show Position	TE	N/A
Select Text	TE	N/A

Client Interface

A client which would like to make an integration using these facilities should consult the following interface:

- A programming interface to communicate with the PostMaster
- A description of the provided services
- A description of the available notifications

External Client types

Clients connecting to the PostMaster must be known by the PostMaster. This can be accomplished in a number of ways:

1. By defining a configuration file containing the extended configuration, name this file to `post.cfd` and to set the environment variable `POSTPATH` to include the directory where the file is stored.

   A change in this file is kind of static nature, since the configuration files are read every time the PostMaster is started.

2. By modifying the configuration dynamically. Requires a PostMaster to be running. The PostMaster provides services for dynamically changing a configuration, adding tools and adding subscriptions. This is easily done via a script and does only affect the current session.

New tool types or message types added to the PostMaster tool and subscription list should be set in a range not conflicting with SDL Suite and TTCN Suite. New tools and messages should use values above `100000`. Exact numbering should follow the scheme used by the SDL Suite and the TTCN Suite:

Item	Description
Tools	Use values in steps of 1000
Service Request	Base the value on the tool providing the service and add local base of 100. Then sequentially number the services
Service Reply	Base the value on the corresponding service request and add 100.

Item	Description
Notifications	Base the value on the tool broadcasting the notification and add the messages sequentially.

Message Based Services

Introduction

Using a message based service requires the client to be connected to the PostMaster. To invoke a service, the application sends a service request message to the tool providing the service. A service request message will normally cause a reply message to be sent to the client. An exception is if the server unexpectedly terminates while processing the service.

The client could choose whether to wait for the reply message (emulating a remote procedure call) or to continue working and intercept the reply message via a callback routine.

The client does not need to subscribe on service request messages or service reply messages. However the client itself must be present in the configuration list.

A server can only process one service at time. If additional services are requested from that particular server when being occupied processing the first service, the server is said to be *busy* and a service error reply is returned to the requester. Other kinds of messages received by server during the processing of a service request are queued up and will be processed as soon as the service has completed.

Service Request

A *service request* normally takes advantage of the `SPSendToTool` function. If more than one instance of a tool is active the function `SPSendToPid` will be more appropriate. The reply is fetched with the `SPRead` function. The client must however verify that the read message is the reply message. The macro `SP_MATCHREPLY` could be used to determine whether the received message is the desired reply.

Service Reply

The first parameter in the *service reply* informs about the result of the service request. The following codes are used:

Status Code	Value	Description
`OK`	`0`	The service was successfully processed. Optional parameters are provided in the reply message.
`Busy`	`1`	The server is *busy* and cannot process the service. The service request is aborted. An additional message might be provided.
`ErrorString`	`2`	The server failed to process the service. The remaining parts of the message contain an error message.
`ErrorCode`	`3`	The server failed to process the service, the next parameter in the service reply is a code indicating the error.

In the detailed description of each service, the reply format is only specified for the normal case returning `OK`. For services replying errors in the `ErrorString` format, the text may be context sensitive and only major error causes are specified.

Error Handling

Error handling takes place at two different levels.

1. **The service request failed to be issued**.

   Service Request message couldn't be sent. In such case the `SPSendToTool`, `SPSendToPid` returns false and the variable `sperrno` indicates the error. It also means that a service reply message will not be sent.

2. **While processing the service**.

   In this case the service request reaches the server but cannot be processed or the processing of the service fails. In such a case, an error code is provided in the service reply message. The error is either provided as a code or as explanatory text.

How to decode the service reply is described in the section below.

Common Errors

These errors are common to all services. They all use the `ErrorString` format. Where applicable, an additional explanatory text is added to the strings below:

```
Bad parameters
Server busy
Service not supported
Server locked
```

SDT Reference Errors

When a service request refers to an SDT reference, the following errors may occur:

```
Reference must contain paranthesis
Reference must start with #SDTREF
Invalid reference
Garbage after reference
Embedded value not terminated
Illegal value in paranthesis after token <token-nr>
Junk after embedded value
Incomplete reference lacking trailing paranthesis
Unsupported reference type <token>
Symbol must be integer >= 0
Malformed coordinate
```

For a reference to the syntax of references, see chapter 18, *SDT References*.

Notifications

Certain services will broadcast notifications as the service is processed in order to inform other participants than the service issuer that an operation has been performed by the server or that the state of the server has changed.

Message Parameters

Normally a message (a service request, a service reply, a notification or other) uses one or more parameters. Generally all such parameters are stored in a PostMaster message's data part.

- Parameters are normally separated by a blank (' ') character.
- A parameter that has an additional '*' character is a shorthand for none or a number of this type, each one separated by a blank ' ' or a newline '\n'.
  - The character '+' might also appear as an alternative. The difference is that one or more is expected.
- A preceding attribute tells how many items to expect.

The following parameter types are used:

Type	Description
`integer`	32 bit integer in ASCII form.
`bool`	Logical. True = 1, False = 0
`string`	ASCII string. If the string contains one or more spaces, it is surrounded by double quotes (i.e. "The string"). If a quote character appears in the string, it is preceded by a backslash ('\'). A backslash character is doubled ('\\'). An empty string is also double quoted ("").
`QString`	ASCII string surrounded by double quotes (i.e. "The string"). If a quote character appears in the string it is preceded by a backslash. ('\').
`ByteString`	Byte string. Is always preceded by an attribute telling the length of the byte string.

Files

- The files `sdt.h` and `itex.h` provides two lists of the available services and notifications in terms of message definitions. It also defines a list (see "Adding Tools and Messages" on page 492 in chapter 10, *The PostMaster*) providing a textual definition of messages.
- The file `sdtsym.h` provides the definitions necessary when working with editors and defines values for diagram types, pages types and symbol types.
- **In Windows**, the file `post.dll` is the DLL that contains the external PostMaster interface. It needs either reside in the same directory as the application that is utilizing it, or in a directory that is in the current `PATH`.

The files are found in the directory `$sdtdir/INCLUDE` **(on UNIX)**, or `%SDTDIR%\include` **(in Windows)**.

Interpretation of a Service Description

Below is an explanation of the different sections found in the service descriptions in "Tool Services" on page 539.

Description

A brief textual description of the service.

Tools Supporting the Service

The tool type(s) providing the service. Corresponds to definitions in the file `sdt.h/itex.h`; see "Files" on page 535.

Service Request

Message name. Corresponds to a definition in the file `sdt.h/itex.h`; see "Files" on page 535.

The service request is presented in a table of service parameters, with the appearance below. The parameters must appear in the order they are listed in the table. Usually, a parameter cannot be omitted. If it is optional, this is explicitly mentioned. See also "Message Parameters" on page 534.

Parameter	Type	Description
`symbolic parameter name`	`parameter type`	Brief description of the parameters.

Service Reply

Message name. Corresponds to a definition in the file `sdt.h/itex.h`; see "Files" on page 535.

The table below is similar to the service request parameter table above. A service reply always contains a status code in the first parameter. Additional parameters are only valid if the status code was OK (0). See also "Message Parameters" on page 534.

Parameter	Type	Description
`status`	`integer`	Service reply status.

Errors

Errors additional to the common errors are listed in this section; see "Common Errors" on page 533.

Emitted Notifications

Notifications emitted as a result of the service being successfully processed; see "Notifications" on page 521.

The Service Encapsulator

The Service Encapsulator application enables access to SDL Suite and TTCN Suite services from the Operating System command line prompt.

Basically the tool connects to the Postmaster, sends a service request message and waits for an answer. When the answer arrives, it is printed on standard output. Finally the application exists.

The Service Encapsulator is also available in source code form to show how the Postmaster's Functional Interface could be used. A description of the internal design of the Service Encapsulator is found in *chapter 12, Using the Public Interface*. The source code of the Service Encapsulator is found in

`$telelogic/examples/public_interface` **(on UNIX)**

`<installation directory>\examples\public_interface` **(in Windows)**

The Service Encapsulator binary is invoked by:

On UNIX: `$telelogic/bin/serverpc <tool> <service> <params>`

In Windows: `<installation directory>\sdtbin\serverpc <tool> <service> <params>`

where <tool> is the tool that should perform the service, and <service> is the service itself.

The `<tool>` and `<service>` arguments could either be entered as a symbolic value or as the assigned integer value. These definitions are found in `sdt.h`

If the service takes parameters, these should be provided in `<params>`

Care should be taken in order to enter parameters correctly. In particular if the service uses quoted string parameters:

- **On UNIX**, surrounding quotes should be doubled since the shell from which the tool is invoked, consumes one pair of quotes.
- **In Windows**, surrounding quotes should be prefixed with a back-slash (\) since the "DOS" shell from which the tool is invoked, consumes the quotes otherwise.

The tool allows carriage return `"\n"` and line-feed `"\r"` to be used.

The tool does not allow <params> to contain binary data. Therefore, the SDL Suite services accepting binary data must only contain ASCII characters.

The tool returns 0 on success and -1 if an error occur. Such errors correspond to errors when calling Postmaster functions, see SPInit, SPSendToTool, SPRead in Functional Interface for possible errors.

Tool Services

Configuration Services

Start Tool

Description

This service will start the specified tool. **On UNIX**, start means that a new UNIX process is “fork’ed” and “exec’ed”. **In Windows**, start means that the applications is started using the Windows API function CreateProcess.

The Service adds “`-post`” to the `argv[1]` variable.

The parameters are recognized by the started tool in the `argv[]` variable starting from index 2.

The started tool inherits the environment used by the PostMaster.

The service behaves somewhat differently from other services since it is performed both in the PostMaster and in the PostMaster library in the started tool. The service is initiated in the PostMaster creating the new tool, but the service is recognized to be completed when the started tool calls the PostMaster function `SPInit`. This call causes a `SESTARTNOTIFY` message to be sent. This message is recognized by the PostMaster which upon reception of the `SESTARTNOTIFY` also sends a `SESTARTREPLY` to the issues of the start service.

The service is also different in its nature since it will cause a time-out to expire if the started tool does not call the `SPInit` function within a certain time limit.

The started tool must have an entry in the PostMaster configuration and the associated file containing the tool to start must exist.

The service will return to the caller:

- When the `SESTARTREPLY` message is received by the caller.
- If the tool is only available in one instance and is already instantiated. In this case the process id of the already instantiated tool is returned.

- If the tool cannot be started or if the tool is not recognized by the PostMaster within a time-out. (The started tool calls `SPInit`.)

The following MSC diagram shows the protocol when a tool is started normally (the Organizer starts the editor).

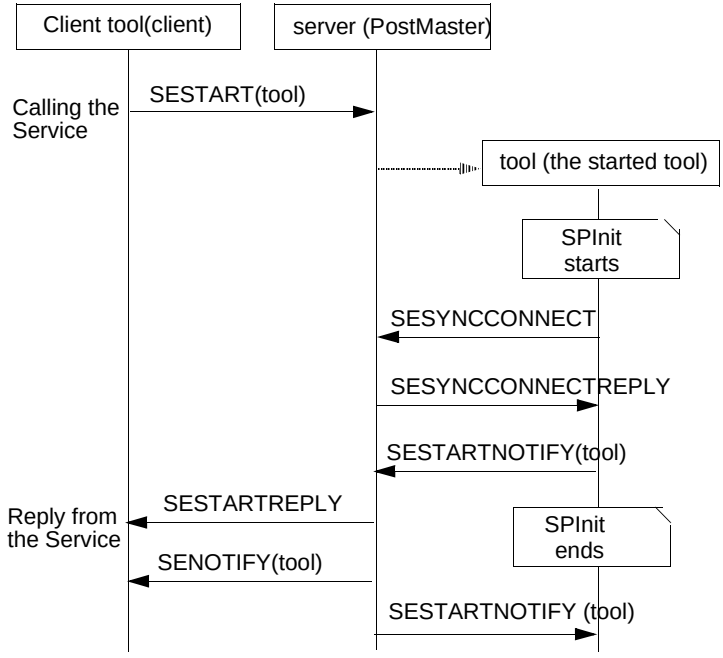


Figure 160: Protocol to start a tool

The SESYNCCONNECT and the SESYNCCONNECTREPLY messages are not present in the **Windows** *implementation.*

Tools Supporting the Service

`SET_POST`

Service Request

SESTART

Parameter	Type	Description
toolType	integer	Tool type to be started
params	string	Optional. Parameters for the started tool. More than one parameter allowed. If separated with blanks, each item will be inserted into the argv[] string list for the started application. The PostMaster will add a parameter -post as the first parameter in the argv[] list.

Service Reply

SESTARTREPLY

Parameter	Type	Description
status	integer	Service reply status.
pId	integer	A PId identifying the tool type of which the corresponding type is started.

Errors

No such tool
Max instances of tool
Already started
File not found
+ Operating System file access and process creation error messages

Emitted Notifications

Notification	Description
SESTARTNOTIFY	Broadcast by the started application when calling SPInit.

Stop Tool

Description

The tool disconnects from the Postmaster and then terminates.

Tools Supporting the Service

`All tools`

Service Request

`SESTOP`

Parameter	Type	Description
`force`	`bool`	If *false*, the tool is allowed to reject the request. If *true*, the tool is expected to terminate

Service Reply

`SESTOPREPLY`

Parameter	Type	Description
`status`	`integer`	Service reply status
`cancelled`	`bool`	*True* if the tool did not accept the Stop request.

Errors

-

Get Tool Type

Description

Returns the process type for a given process id.

Tools Supporting the Service

SET_POST

Service Request

SEGETTOOLTYPE

Parameter	Type	Description
pId	integer	A PId identifying the tool type of which the corresponding type is wanted.

Service Reply

SEGETTOOLTYPEREPLY

Parameter	Type	Description
status	integer	Service reply status.
noOfToolType	integer	Number of tools corresponding to the PId value. If the requested tool type was not found in the configuration, noOfToolType is set to 0.
toolType	integer	The type of tool. If the requested PId is not found, toolType 0 is returned

Errors

-

Get Tool Pid

Description

Returns the process id for the requested tool type. If multiple instances of the tool type exist, all PId values are returned.

Tools Supporting the Service

SET_POST

Service Request

SEGETTOOLPID

Parameter	Type	Description
toolType	integer	An identifier for the tool type of which the corresponding PIds) are wanted.

Service Reply

SEGETTOOLPIDREPLY

Parameter	Type	Description
status	integer	Service reply status.
noOfPid	integer	Number of PId values corresponding to the toolType. If the requested tool type was not found in the configuration, noOfPid is set to 0.
pId	integer	A PId value corresponding to toolType

Errors

-

Add Tool

Description

Dynamically adds a tool type to the PostMaster configuration.

Tools Supporting the Service

SET_POST

Service Request

SEADDTOOL

Parameter	Type	Description
toolType	integer	An identifier for the tool. Should be a value within the allowed range.
filename	string	The filename of the executable tool. Should be a pure filename and **must not** include a directory path. The file should be stored on a directory that is included in the directories designated by the environment variable POSTPATH.

Service Reply

SEADDTOOLREPLY

Parameter	Type	Description
status	integer	Service reply status.
exist	bool	Returns false if the tool was inserted, true if it already existed.

Errors

Operation failed

Add Tool Subscription

Description

Dynamically adds a message to a tool's subscription list.

Tools Supporting the Service

`SET_POST`

Service Request

`SEADDTOOLSUBSCRIPTION`

Parameter	Type	Description
`tooltype`	`integer`	The tool type to which a message subscription should be added.
`message`	`integer`	The message to add to the subscription list.

Service Reply

`SEADDTOOLSUBSCRIPTIONREPLY`

Parameter	Type	Description
`status`	`integer`	Service reply status.
`exist`	`bool`	Returns `false` if the message was inserted, `true` if it already existed.

Errors

```
No such tool
Operation failed
```

System File Services

List System Files

Description

Lists the files currently held in the Diagram Structure chapter in the Organizer. The files will be returned in the order they are found in the Organizer (i.e. top-down, left-right).

Tools Supporting the Service

SET_ORGANIZER

Service Request

SELISTSYSTEMFILES

Parameter	Type	Description
-	-	N/A.

Service Reply

SELISTSYSTEMFILESREPLY

Parameter	Type	Description
status	integer	Service reply status.
noOfFiles	integer	Number of returned files.
file(s)	string	A list of files, with a complete directory path. Each file specification is ended by a newline.

Errors

-

New System

Description

Clear the Organizer content and create a new system. Corresponds to the Organizer menu choice *New*.

Tools Supporting the Service

`SET_ORGANIZER`

Service Request

`SENEWSYSTEM`

Parameter	Type	Description
`forceQuit`	`bool`	If *true*, quit modified diagrams. If *false* and there were one or more modified diagrams, the service is denied.

Service Reply

`SENEWSYSTEMREPLY`

Parameter	Type	Description
`status`	`integer`	Service reply status.
`cancelled`	`bool`	*True*, if the service was cancelled.

Errors

```
Service request denied. No license available.
```

Open System

Description

Opens a system file and displays the file contents in the Organizer. Corresponds to the Organizer menu command *Open*.

Tools Supporting the Service

`SET_ORGANIZER`

Service Request

`SEOPENSYSTEM`

Parameter	Type	Description
`filename`	`string`	The system file to open.
`forceQuit`	`bool`	If `true`, quit modified diagrams. If `false` and modified diagram(s) exist(s), the service is denied.

Service Reply

`SEOPENREPLY`

Parameter	Type	Description
`status`	`integer`	Service reply status.
`toolType`	`integer`	The type of tool. If the requested PId is not found, `toolType` 0 is returned.

Errors

```
Service request denied. No license available
Cannot open, system modified
Error opening system A message box displays the
error
```

Save System

Description

Corresponds to the Organizer menu command *Save*. Unconnected diagrams are saved in default file names; for more information, see "Save in file" on page 63 in chapter 2, *The Organizer*.

Tools Supporting the Service

`SET_ORGANIZER`

Service Request

`SESAVEALL`

Parameter	Type	Description
`systemStructureOnly`	`bool`	True, both the system files and diagram files are saved; False, only diagram files are saved.

Service Reply

`SESAVEALLREPLY`

Parameter	Type	Description
`status`	`integer`	Service reply status.

Errors

```
Service request denied. No license available
Error save all A message box displays the error
```

Add Existing

Description

Adds an existing document to the Organizer and optionally displays it in an editor. The type of document to be added and its corresponding editor depends on the extension of the given filename. This is the same mechanism as in the GUI-based equivalent. Corresponds to the Organizer menu choice *Add Existing*.

Tools Supporting the Service

`SET_ORGANIZER`

Service Request

`SEADDEXISTING`

Parameter	Type	Description
`filename`	`string`	The document to add in the Organizer.
`selected Filename`	`string`	If there is a document connected to this file in the Organizer, the document will be selected before the *Add Existing* operation. In this case, the added document will be inserted before the selected document. An empty string means no selection.
`start Editor`	`bool`	If *true*, the added document will be popped up in an editor.
`expand Substructure`	`bool`	If *true*, the substructure diagrams to the added SDL diagram are added as well.

Service Reply

`SEADDEXISTINGREPLY`

Parameter	Type	Description
`status`	`integer`	Service reply status.
`ok`	`bool`	Returns *true* if the operation could be performed.

Errors

-

Link File Services

Add Local Link File

Description

Prepares a personal link file for a user. Any changes in endpoint and link information after this service is called are saved into this file. The master link file can be read only for the user at this time.

Tools Supporting the Service

SET_ORGANIZER

Service Request

SEADDLOCALLINKFILE

Parameter	Type	Description
filename	string	File name.

Service Reply

SEADDLOCALLINKFILEREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

-

Merge Local Link File

Description

Merges the changes saved into the personal link file into the master link file. After a successful merge, the local link file information is cleared.

Tools Supporting the Service

SET_ORGANIZER

Service Request

SEMERGELOCALLINKFILE

Parameter	Type	Description
-	-	N/A.

Service Reply

SEMERGELOCALLINKFILEREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

-

TTCN Suite Services

Convert to GR

Description

Converts a MP file to TTCN-GR. One document file is generated in the specified destination directory. The document filename is generated by the TTCN Suite. The name of the destination directory is given. The parameter *ignore page number* indicates that any included page numbers in the MP file will be ignored.

The generated names are displayed in the Organizer log. For more information, see "Importing a TTCN-MP Document" on page 1157 in chapter 24, *The TTCN Browser (on UNIX)*.

Tools Supporting the Service

`IET_BASE`

Service Request

`IEBXCONVERTTOGR`

Parameter	Type	Description
`filename`	`QString`	The name of the MP file
`destination`	`QString`	The name of the destination directory
`ignore page number`	`bool`	Ignoring the included page numbers

Service Reply

`IEBXCONVERTTOGRREPLY`

Parameter	Type	Description
`status`	`integer`	Service reply status
`itexfile`	`QString`	The filename of the generated document.

Errors

```
The MP file does not exist
Illegal MP file
The file has no read permission
Unable to generate document files in the destination
```

Load Document

Description

Given a source file path this function load the document. Then this document may be shown in the TTCN Browser using "Show" service. The source file may have MP format or TTCN GR format.

Note

Available on Windows only.

Tools Supporting the Service

`IET_BASE`

Service Request

`SELOAD`

Parameter	Type	Description
`filename`	`QString`	The filename

Service Reply

SELOADREPLY

Parameter	Type	Description
`status`	`integer`	Service reply status (zero if the operation does not fail).
`buffid`	`integer`	The document's buffer identifier

Errors

```
No such file or directory
Syntax error
```

Show Document

Description

Show the given document in the TTCN Browser.

Tools Supporting the Service

IET_BASE

Service Request

SESHOW

Parameter	Type	Description
buffid	integer	The document's buffer identifier

Service Reply

SESHOWREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).

Errors

Invalid buffer identifier

Opened Documents

Description

Retrieves information about open documents known to TTCN Suite. The EBNF follows:

```
DocInfoList ::= empty | DocInfoList '<NewLine>'
                DocInfo
DocInfo     ::= documentidentifier ':' buffid ':'
                itexfile ':' backupfile
```

Tools Supporting the Service

IET_BASE

Service Request

IEBXOPENEDDOCUMENTS

Service Reply

IEBXOPENEDDOCUMENTSREPLY

Parameter	Type	Description
`status`	`integer`	Service reply status (zero if the operation does not fail).
`infolist`	`EBNF`	The list of open documents identifiers, buffer identifiers and database file.

Errors

-

Fetch Buffer Identifier Given the Database

Description

Given a database, this function fetches the document buffer identifier if the document is open.

Tools Supporting the Service

IET_BASE

Service Request

IEBXGETBUFFIDFROMPATH

Parameter	Type	Description
itexfile	QString	The database path in the local system syntax.

Service Reply

IEBXGETBUFFIDFROMPATHREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).
buffid	integer	The document's buffer identifier

Errors

Unable to find database

Fetch Buffer Identifier Given MP File Path

Description

Given a MP file path this function fetches the document buffer identifier if there is any database generated from this MP and the document in the database is open.

Tools Supporting the Service

IET_BASE

Service Request

IEBXGETBUFFIDFROMMPPATH

Parameter	Type	Description
mpfile	QString	The MP file path in the local system syntax.

Service Reply

IEBXGETBUFFIDFROMMPPATHREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).
buffid	integer	The document's buffer identifier

Errors

Unable to find database

Convert to MP

Description

Generates a MP file for the given system node. This converts the entire document source (all TTCN objects) and does not consider the selection. The document must be connected but not necessarily open.

For more information, see "Exporting a TTCN Document to TTCN-MP" on page 1152 in chapter 24, *The TTCN Browser (on UNIX)*.

Tools Supporting the Service

```
IET_BASE
```

Service Request

```
IEBXCONVERTTOMP
```

Parameter	Type	Description
buffid	integer	The document's buffer identifier
standard flag	bool	Indicates if the MP shall be TTCN standard. (True if standard MP is required.)
filename	QString	The MP filename

Service Reply

```
IEBXCONVERTTOMPREPLY
```

Parameter	Type	Description
status	integer	Service reply status

Errors

```
Invalid buffer identifier
The document is not connected
Unable to write file
```

Convert Selection to MP

Description

Generates a MP file for the given system node. It converts only the selected parts of the document which means that the document needs both to be connected and open.

For more information, see "Exporting a TTCN Document to TTCN-MP" on page 1152 in chapter 24, *The TTCN Browser (on UNIX)*.

Tools Supporting the Service

`IET_BASE`

Service Request

`IEBXCONVERTSELTOMP`

Parameter	Type	Description
`buffid`	`integer`	The document's buffer identifier
`standard flag`	`bool`	Indicates if the MP shall be TTCN standard. (True if standard MP is required.)
`filename`	`QString`	The MP filename

Service Reply

`IEBXCONVERTSELTOMPREPLY`

Parameter	Type	Description
`status`	`integer`	Service reply status

Errors

```
Invalid buffer identifier
The document is not connected
The document is not open
Unable to write file
```

Merge Document

Description

This function is used to merge one document into another document. Both the source and the destination documents must be available to the TTCN Suite environment, i.e. both documents must be loaded. Furthermore a selection must exists in the source document.

For more information, see "Merging TTCN Documents" on page 1144 in chapter 24, *The TTCN Browser (on UNIX)*.

Tools Supporting the Service

IET_BASE

Service Request

IEBXMERGEDOCUMENT

Parameter	Type	Description
srcbuffid	integer	The source document's buffer identifier
destbuffid	integer	The destination document's buffer identifier

Service Reply

IEBXMERGEDOCUMENTREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).

Errors

Unable to merge documents

Analyze Document

Description

Analyze the given system node.

For more information, see chapter 26, *Analyzing TTCN Documents (on UNIX)*, or chapter 31, *Analyzing TTCN Documents (in Windows)*.

Tools Supporting the Service

IET_BASE

Service Request

IEBXANALYZE

Parameter	Type	Description
buffid	integer	The document's buffer identifier
forced analysis	bool	Set if forced analysis should be in effect
verbose	bool	Indicates if verbose mode is to be on
errorlimit	integer	Max number of errors before aborting

Service Reply

IEBXANALYZEREPLY

Parameter	Type	Description
status	integer	Service reply status

Errors

Invalid buffer identifier
The document is not connected

Close Document

Description

This function closes the document and all open tables in it.

Tools Supporting the Service

IET_BASE

Service Request

IEBXCLOSEDOCUMENT

Parameter	Type	Description
buffid	integer	The document's buffer identifier

Service Reply

IEBXCLOSEDOCUMENTREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).

Errors

Invalid buffer identifier
Unable to close document

Save Document

Description

Given the buffer identifier of the document the corresponding system node is saved in the given filename.

Tools Supporting the Service

IET_BASE

Service Request

IEBXSAVE

Parameter	Type	Description
buffid	integer	The document reference
filename	QString	The filename where the document source must be saved.

Service Reply

IEBXSAVEREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

```
No such tool
Invalid buffer identifier
Couldn't write file
```

Selector

Description

Given restrictions and a database this function selects all objects which fulfill the restrictions. For more information, see "Using More Complex Selections" on page 1124 in chapter 24, *The TTCN Browser (on UNIX)*.

Note

Available on UNIX only.

Tools Supporting the Service

`IET_BASE`

Service Request

`IEBXSELECTOR`

Parameter	Type	Description
`buffid`	`integer`	The document's buffer identifier
`namerestr`	`string`	The name restriction.
`typerestr`	`string`	The content restriction.
`contentrestr`	`string`	The type restriction.
`selectormode`	`string`	The selector mode (restrict, extend or replace REL where REL is references, references_recursive or referenced_by)
`analysestatus`	`string`	The analysis status (not_analyzed, error_analyzed or ok_analyzed)

Service Reply

`IEBXSELECTORREPLY`

Parameter	Type	Description
`status`	`integer`	Service reply status (zero if the operation does not fail).

Errors

`No such tool`

```
Invalid buffer identifier
```

SelectAll

Description

Select all objects in the document.

Tools Supporting the Service

```
IET_BASE
```

Service Request

```
IEBXSELECTALL
```

Parameter	Type	Description
buffid	integer	The document's buffer identifier

Service Reply

```
IEBXSELECTALLREPLY
```

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).

Errors

```
Invalid buffer identifier
```

DeselectAll

Description

Remove all selections in the document.

Tools Supporting the Service

IET_BASE

Service Request

IEBXDESELECTALL

Parameter	Type	Description
buffid	integer	The document's buffer identifier

Service Reply

IEBXDESELECTALLREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).

Errors

Invalid buffer identifier

IsSelected

Description

This function checks if there is any object selected in the document.

Tools Supporting the Service

IET_BASE

Service Request

IEBXISSELECTED

Parameter	Type	Description
buffid	integer	The document's buffer identifier

Service Reply

IEBXISSELECTEDREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).
selectstate	bool	Indicates if there is any selected objects in the document.

Errors

Invalid buffer identifier

Get Modify Time

Description

This function fetches the modify time of a document.

Tools Supporting the Service

IET_BASE

Service Request

IEBXGETDOCUMENTMODIFYTIME

Parameter	Type	Description
buffid	integer	The document's buffer identifier

Service Reply

IEBXGETDOCUMENTMODIFYTIMEREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).
modifytime	QString	The modify time of the document.

Errors

Invalid buffer identifier

Get Path

Description

Given a document buffer identifier this function fetches the corresponding database path.

Tools Supporting the Service

IET_BASE

Service Request

IEBXGETPATH

Parameter	Type	Description
buffid	integer	The document's buffer identifier

Service Reply

IEBXGETPATHREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).
itexfile	QString	The database path in the local system syntax.

Errors

Invalid buffer identifier

Get MP Path

Description

Given a document bufferid this function fetches the corresponding MP file path.

Tools Supporting the Service

IET_BASE

Service Request

IEBXGETMPPATH

Parameter	Type	Description
buffid	integer	The document's buffer identifier

Service Reply

IEBXGETMPPATHREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).
mpfile	QString	The MP file path in the local system syntax.

Errors

Invalid buffer identifier

Find Table

Description

Given the buffer identifier of the selected document and a table identifier, TTCN Suite searches for the table. The found table is displayed in the Table Editor.

Tools Supporting the Service

IET_BASE

Service Request

IEBXFINDTABLE

Parameter	Type	Description
buffid	integer	The document's buffer identifier
tableid	QString	The name of the table to find

Service Reply

IEBXFINDTABLEREPLY

Parameter	Type	Description
status	integer	Service reply status

Errors

Invalid buffer identifier
Unconnected document

Close Table

Description

Close the given table.

Tools Supporting the Service

IET_BASE

Service Request

IEBXCLOSETABLE

Parameter	Type	Description
buffid	integer	The document's buffer identifier
tableident	QString	The table identifier.

Service Reply

IEBXCLOSETABLEREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).

Errors

```
Invalid buffer identifier
No such object
```

Get Table State

Description

This function returns the status of a given table. The status a table indicates if the table is open or close.

Tools Supporting the Service

IET_BASE

Service Request

IEBXGETTABLESTATE

Parameter	Type	Description
buffid	integer	The document's buffer identifier
tableident	QString	The table identifier.

Service Reply

IEBXGETTABLESTATEREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).
tablestate	QString	Returns open, close, row_in_open or row_in_close.

Errors

Invalid buffer identifier
No such object

Get Row Number

Description

Given the name of a row this function returns the number (position) of it in the table. The row can be a row in a single or multiple table.

Tools Supporting the Service

IET_BASE

Service Request

IEBXGETROWNUMBER

Parameter	Type	Description
buffid	integer	The document's buffer identifier
tableident	QString	The table identifier.
rowname	identifier	The name of the row.

Service Reply

IEBXGETROWNUMBERREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).
rownumber	integer	The number of the given row. The first row has number one.

Errors

```
Invalid buffer identifier
No such object
No such row
```

Select Row

Description

This function modifies the selection status of a given row in a table.

Tools Supporting the Service

IET_BASE

Service Request

IEBXSELECTROW

Parameter	Type	Description
buffid	integer	The document's buffer identifier
tableident	QString	The table identifier.
rownumber	integer	The number of the row.
selectstate	bool	The select status of the row to be modified.

Service Reply

IEBXSELECTROWREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).
exist	bool	Returns false if the message was inserted, true if it already existed.

Errors

```
Invalid buffer identifier
No such object
No such rownumber
```

Clear Selection

Description

This function removes all row selections in a table.

Tools Supporting the Service

IET_BASE

Service Request

IEBXCLEARSELECTION

Parameter	Type	Description
buffid	integer	The document's buffer identifier
tableident	QString	The table identifier.

Service Reply

IEBXCLEARSELECTIONREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).

Errors

Invalid buffer identifier
No such object

Rows Selected

Description

Given the buffer identifier and the table identifier, this function returns the numbers of the selected rows in a table (the first row has number 1).

Tools Supporting the Service

IET_BASE

Service Request

IEBXROWSSELECTED

Parameter	Type	Description
buffid	integer	The document's buffer identifier
tableident	QString	The table identifier.

Service Reply

IEBXROWSSELECTEDREPLY

Parameter	Type	Description
status	integer	Service reply status (zero if the operation does not fail).
nonezero*	integer	A list of selected row numbers. The list is empty if no row is selected.

Errors

Invalid buffer identifier
No such object

Menu Manipulation Services

Introduction

All graphical tools in the SDL Suite support customizable menus. These user-defined menus will be appended to the menu bar of the tool. The exact location will be defined by the tool, depending on the abilities of the graphical framework the tool is built upon.

The intention is to have an external tool to configure a tool in order to provide the necessary UI. SDL Suite and TTCN Suite could also take advantage of these extendable menus.

The following operations are available through a set of well defined PostMaster messages:

- Creating a menu
- Creating a menu choice
- Deleting a menu

Each menu choice can be associated to any of the following:

- A PostMaster message
- An operating system command

Add Menu

Description

Adds a new menu to the menu bar.

Tools Supporting the Service

SET_ORGANIZER
SET_SDLE
SET_MSCE
SET_OME
SET_TE
SET_SIMULATOR_UI [1]
SET_FILEVIEWER
SET_COVERAGEVIEWER
SET_XREFVIEWER
SET_TYPEVIEWER
SET_TREEVIEWER
SET_PREFERENCES
SET_HELPVIEWER

Service Request

SEMENUADD

Parameter	Type	Description
menuName	string	Name of the menu to add.

Service Reply

SEMENUADDREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

-

1. **In Windows**, the SDL Simulator/Explorer UI's can **not** read menu-definition files.

Delete Menu

Description

Removes the menu and its menu choices from the menu bar.

Tools Supporting the Service

SET_ORGANIZER
SET_SDLE
SET_MSCE
SET_OME
SET_TE
SET_SIMULATOR_UI [1]
SET_FILEVIEWER
SET_COVERAGEVIEWER
SET_XREFVIEWER
SET_TYPEVIEWER
SET_TREEVIEWER
SET_PREFERENCES
SET_HELPVIEWER

Service Request

SEMENUDELETE

Parameter	Type	Description
menuName	string	Name of the menu to delete.

Service Reply

SEMENUDELETEREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

-

1. **In Windows**, the SDL Simulator/Explorer UI's can **not** read menu-definition files.

Clear Menu

Description

Clears the menu bar from a menu item.

Tools Supporting the Service

SET_ORGANIZER
SET_SDLE
SET_MSCE
SET_OME
SET_TE
SET_SIMULATOR_UI [1]
SET_FILEVIEWER
SET_COVERAGEVIEWER
SET_XREFVIEWER
SET_TYPEVIEWER
SET_TREEVIEWER
SET_PREFERENCES
SET_HELPVIEWER

Service Request

SEMENUCLEAR

Parameter	Type	Description
menuName	string	Name of the menu to clear.

Service Reply

SEMENUCLEARREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

-

1. **In Windows**, the SDL Simulator/Explorer UI's can **not** read menu-definition files.

Add Item to Menu

Description

Adds a menu choice to the specified menu. The menu choice could either perform an OS command or issue a PostMaster notification when selected. The OS command to perform or the message to broadcast could be sensitive on a selected symbol.

The description of the service parameters below is generic to all tools supporting the service. Some tools have special interpretations of some parameters. Some tools also allow *format codes* to be used in the command string or as message parameter, providing additional context sensitive information. Both these tool-specific issues are described separately in the following sections:

- "Add Item to Menu – Organizer" on page 588
- "Add Item to Menu – Text Editor" on page 592
- "Add Item to Menu – Graphical Editors" on page 593

Tools Supporting the Service

```
SET_ORGANIZER
SET_SDLE
SET_MSCE
SET_OME
SET_TE
SET_SIMULATOR_UI [1]
SET_FILEVIEWER
SET_COVERAGEVIEWER
SET_XREFVIEWER
SET_TYPEVIEWER
SET_TREEVIEWER
SET_PREFERENCES
SET_HELPVIEWER
```

1. **In Windows**, the SDL Simulator/Explorer UI's can **not** read menu-definition files.

Service Request

SEMENUADDITEM

Parameter	Type	Description
menuName	string	The menu in which an item should be added.
menuItem	string	Name of menu item to add.
separator	bool	If a separator should precede the item.
statusText	string	The status text to show when the menu item is selected.
notUsed1 lastAction ProprietaryKey (tool-specific)	integer	The interpretation of this parameter is tool-specific; see the separate tool descriptions later. If not used, this parameter should be set to 0.
notUsed2 AttributeKey (tool-specific)	integer	The interpretation of this parameter is tool-specific; see the separate tool descriptions later. If not used, this parameter should be set to 0.
scope	integer	Indicates when the menu item should be dimmed. The possible values are tool-specific; see the separate tool descriptions later. If not used, this parameter should be set to:
		ALWAYS (0) The menu choice will always be available, independently of the selection in the tool's active window.

Parameter	Type	Description
`confirmText`	`string`	If no text is provided, no confirmation is assumed. A non-empty text denotes confirmation; a two button dialog will be issued with the choices *OK* and *Cancel*. The dialog text is defined in `confirmText`. In an associated user editable field, the expanded action to perform is displayed.
`actionType`	`integer`	The value controls the last part of the message which is variant. 0 = PostMaster message 1 = OS command.

- If `actionType` above sets the variant part to be a **PostMaster message**, the last two parameters are:

Parameter	Type	Description
`message`	`integer`	Interpreted as the PostMaster message to send.
`params`	`string`	Interpreted as the parameters to the PostMaster message. Tool-specific context sensitive format codes are evaluated; see the separate tool descriptions later.

- If `actionType` above sets the variant part to be an **OS command**, the last two parameters are:

Parameter	Type	Description
`OSblock`	`bool`	Whether to wait for the command to finish before giving control to the user.

Parameter	Type	Description
OScommand	string	The OS command to perform. Tool-specific context sensitive format codes are evaluated; see the separate tool descriptions later.

Service Reply

SEMENUADDITEMREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

-

Add Item to Menu – Organizer

Description

Identical to the description of *Add Item to Menu*, except that the semantics of the parameters and supported format codes for the service differ.

Tools Supporting the Service

`SET_ORGANIZER`

Service Request

`SEMENUADDITEM`

Format Codes

The following format codes are recognized. There are format modifiers for some of the basic formats:

- B – the base filename
- D – the directory part of the full filename
- R – the filename, relative to the *source directory*

Format code	Description
`%b` `%Bb,` `%Db` `%Rb`	The file name of the system file associated to the Organizer window.
`%f` `%Bf` `%Df` `%Rf`	The name of the file that contains the selected object.
`%F` `%BF` `%DF` `%RF`	All files in the substructure of the selected object (including the selected object).
`%l` `%Bl` `%Dl` `%Rl`	The name of the link file loaded in the Organizer.
`%o` `%Bo` `%Do` `%Ro`	The name of the Control Unit file for the selected object.

Format code	Description
%r	Perform the command recursively on all files in the substructure of the selected object (including the selected object).
%u	The source directory.
%v	The target directory.

Parameters

Apart from providing a set of format codes, the Organizer gives special interpretation to the lastAction and scope parameters.

Parameter	Type	Description
lastAction	integer	Controls what happens when a dynamic menu command has been performed. Only available when actionType is OS Command. This attribute should be one of the following:
		NOTHING (0) Perform no action.
		CHECK_FILE (1) A check file operation is performed on the selected object when the OS command is completed. If the OS command is non-blocking, the command is performed as soon as the OS command has been issued.
		CHECK_FILE_ON_RECURSIVE (2) This flag works as the CHECK_FILE flag but is used only with a %r command.

Parameter	Type	Description
		`CM_UPDATE (4)` Perform a CM_UPDATE operation on the closest *.scu file.
		`SHOW_LOG (8)` Show the Organizer Log window (where textual output from the OS command is presented).

Parameter	Type	Description
scope	integer	This attribute could be one of the following:
		ALWAYS (0) The menu choice will always be available, independently of the selection in the tool's active window.
		ONE_SELECTED_OBJECT (1) The menu choice is available if one object is selected.
		SELECTED_OBJECT_NOT_IN_EDITOR (4) The menu choice is available if one object is selected and the selected object is not loaded in an editor.
		VALID_SELECTED_OBJECT (5) The menu choice is available if one object is selected and the object is not marked invalid.
		SELECTED_GROUP_NOT_IN_EDITOR (6) The menu choice is available if one object is selected and **no diagram** of the document group for the selected object is loaded in an editor.

Add Item to Menu – Text Editor

Description

Identical to the description of *Add Item to Menu*, except that the semantics of the parameters and supported format codes for the service differ.

Tools Supporting the Service

```
SET_TE
```

Service Request

```
SEMENUADDITEM
```

Format Codes

The following format codes are recognized.

Format code	Description
`%f`	The filename of the document.
`%u`	The source directory
`%s`	The selected text
`%S`	The entire text of the document.

Add Item to Menu – Graphical Editors

Description

Identical to the description of *Add Item to Menu*, except that the semantic of the parameters and supported format codes for the service differ.

Tools Supporting the Service

```
SET_SDLE
SET_MSCE
SET_OME
```

Service Request

```
SEMENUADDITEM
```

Format Codes

The following format codes are recognized.

Format code	Description
`%a`	The absolute name of the file associated with the selected object. Extracted from the SDT reference.
`%b`	The absolute name of the file associated to the window.
`%c`	If the selected object uses extended data, the `comment` is extracted.
`%d`	If the selected object uses extended data, the data part is extracted.
`%e`	The text in the object. (Only in the SDL Editor.)
`%f`	The name of the file that contains the object.
`%g`	The SDT reference to the object.
`%p`	The page name currently shown in the window (not applicable to the MSC Editor).
`%s`	The name of the file shown in the window.
`%t`	The page name for the object (not applicable to the MSC Editor). Extracted from the SDT reference.

Parameters

Apart from providing a set of format codes, the graphical editors give special interpretation to the `lastAction`, `ProprietaryKey` and `AttributeKey` parameters.

Parameter	Type	Description
`ProprietaryKey`	`integer`	The keys `ProprietaryKey` and `AttributeKey` are used to determine whether or not a menu choice should be available (i.e. dimmed or not). See "Editor – Object Services" on page 610 for more information.
`AttributeKey`	`integer`	See description of `ProprietaryKey`.

Parameter	Type	Description
scope	integer	This attribute should be one of the following:
		ALWAYS (0) The menu choice will always be available, independently of the selection in the tool's active window. The keys ProprietaryKey and AttributeKey are handled as don't care.
		ONE_SELECTED_OBJECT[a] (1) The menu choice is available only if exactly one object is selected. The keys ProprietaryKey and AttributeKey are handled as don't care.
		MATCHING_KEYS (2) The menu choice is available only if at least one of the selected objects has an attribute that matches the keys above.
		MATCHING_KEYS_ONE_SELECTED_OBJECT (3)[a.] The menu choice is available only if exactly one object is selected and it has an attribute that matches the keys above.

a. Each tool should define and adopt conventions for when exactly one object only is considered as selected. For instance, selecting a task symbol in an SDL Editor also selects the from and to lines. However, attaching information to these lines does not fill any meaningful purpose; the SDL Editor considers the situation as if one object only (i.e. the task symbol) was selected.

Logging Services

Start MSC Log

Description

Starts the logging of messages sent by tools connected to the PostMaster. The format used by the log is event-oriented MSC/PR.

Tools Supporting the Service

SET_POST

Service Request

SESTARTTRACE

Parameter	Type	Description
filename	string	The name of the file on which the log should be stored. If the parameter is omitted, the default log file post.mpr will be used. If the parameter is set to "-e", logging is done on standard error.
logMode	integer	If not set, only public messages are logged. If set to a value > 2 all messages are logged.

Service Reply

SESTARTTRACEREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

-

Stop MSC Log

Description

Stops the logging of messages sent by PostMaster tools.

Tools Supporting the Service

SET_POST

Service Request

SESTOPTRACE

Parameter	Type	Description
-	-	N/A.

Service Reply

SESTOPTRACEREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

-

SDT Reference Services

Show Source

Description

Selects the SDT reference in an editor. A reference could be:

- An SDL reference.

  The reference is shown in an SDL Editor.

- An MSC reference.

  The reference is shown in an MSC Editor.

- An OM reference.

  The reference is shown in an OM Editor.

- A text reference

  The reference is shown in a text editor. The text editor is chosen by the preference SDT*TextEditor.

For a complete description of the format of an SDT reference, please see chapter 18, *SDT References*.

Tools Supporting the Service

`SET_ORGANIZER`

Service Request

`SESHOWREF`

Parameter	Type	Description
`SDTRef`	`string`	A valid SDT reference.

Service Reply

`SESHOWREFREPLY`

Parameter	Type	Description
`status`	`integer`	Service reply status.

Errors

`SDT Reference Errors`

Emitted Notifications

Notification	Description
SELOADNOTIFY	If any editor loaded the required diagram.
SESDLELOADNOTIFY	If the SDL Editor loaded the required diagram.
SEOMELOADNOTIFY	If the OM editor loaded the required diagram.
SESTARTNOTIFY	If the editor was started.

Obtain GR Reference

Description

Returns the SDT references for the current selection(s) in the editors.

Tools Supporting the Service

SET_SDLE
SET_MSCE
SET_OME

Service Request

SEOBTAINGRREF

Parameter	Type	Description
-	-	N/A

Service Reply

SEOBTAINGRREFREPLY

Parameter	Type	Description
status	integer	Service reply status.
NoOfRef	integer	The number of references found.
ref*	string	A complete SDT reference.

Errors

SDT Reference Errors
Illegal use of qualifier
Illegal reference type

Editor – Diagram Services

The Buffer Concept

The editor defines the concept buffer, which basically identifies a diagram currently loaded in the editor. Each buffer is identified with a *buffer id* which is unique within one session of the editor as long as the editor is not stopped)

A buffer id is returned when an existing diagram is successfully loaded in an editor or a new diagram is created and implicitly loaded in the editor. Then, most services manipulating diagrams in editors, refer to the buffer containing the diagram via the buffer id.

Load Diagram

Description

Loads a diagram or text document specified by filename in an editor buffer. If the diagram was already loaded, the existing buffer id will be returned.

Tools Supporting the Service

SET_SDLE
SET_MSCE
SET_OME
SET_TE

Service Request

SELOAD

Parameter	Type	Description
filename	string	The diagram file to load specified with the full directory path.

Service Reply

SELOADREPLY

Parameter	Type	Description
status	integer	Service reply status.
bufId	integer	Refers to an allocated buffer in the editor.
type	integer	The type of diagram.
name	string	The name of the loaded diagram.

Errors

Can not read file <filename> <error message>
SDLE is busy, syntax error in text *(SDLE only)*

Emitted Notifications

Notification	Description
SELOADNOTIFY	If the diagram was loaded.
SESDLELOADNOTIFY	If an SDL diagram was loaded.
SEOMELOADNOTIFY	If an OM, SC or HMSC diagram was loaded.

Unload Diagram

Description

Unloads a diagram from an editor.

Tools Supporting the Service

SET_SDLE
SET_MSCE
SET_OME
SET_TE

Service Request

SEUNLOAD

Parameter	Type	Description
bufId	integer	Refers to a buffer in the editor.
forceUnload	bool	If true, the editor will force a modified diagram to be unloaded. If false, the editor will not unload the diagram if it is modified.

Service Reply

SEUNLOADREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

Invalid diagram buffer id
Diagram is changed *If force Unload is false*
SDLE is busy, syntax error in text *(SDLE only)*

Emitted Notifications

Notification	Description
SEUNLOADNOTIFY	If the diagram was unloaded.

Show Diagram

Description

The service will raise a window in the editor showing the specified buffer.

Tools Supporting the Service

```
SET_SDLE
SET_MSCE
SET_OME
SET_TE
```

Service Request

```
SESHOW
```

Parameter	Type	Description
bufId	integer	Refers to a buffer in the editor.
pageName	string	Name of the page to show. If the string is empty, the last recently used, or if no such page exist, the default page will be shown. Applicable all diagrams except MSC and text documents. For MSC diagrams this parameter may be omitted or left empty. For text document (loaded in the TE), this option is ignored.

Service Reply

```
SESHOWREPLY
```

Parameter	Type	Description
status	integer	Service reply status.

Errors

```
Invalid diagram buffer id
Unable to open page   (SDLE/OME only)
SDLE is busy, syntax error in text   (SDLE only)
```

Emitted Notifications

Notification	Description
SESHOWNOTIFY	If the diagram was raised.

Save Diagram

Description

The service will save the diagram in the specified file. If not a valid filename or if there is no permission to save the file, an error will be returned. When saved, the editor buffer is marked as not edited.

Tools Supporting the Service

```
SET_SDLE
SET_MSCE
SET_OME
SET_TE
```

Service Request

```
SESAVE
```

Parameter	Type	Description
bufId	integer	Refers to a buffer in the editor.
filename	string	The file where to save the buffer.

Service Reply

```
SESAVEREPLY
```

Parameter	Type	Description
status	integer	Service reply status.
saveok	bool	Returns `true` if the buffer was successfully saved.

Errors

```
Invalid diagram buffer Id
Diagram is new. Filename is missing.
Cannot save file <filename> <error message>
SDLE is busy, syntax error in text    (SDLE only)
```

Emitted Notifications

Notification	Description
SESAVENOTIFY	If the diagram was saved.

Create SDL Diagram

Description

Creates an empty SDL diagram in a new buffer. The diagram gets an unconnected status. Corresponds to SDL Editor command *New*.

Tools Supporting the Service

`SET_SDLE`

Service Request

`SESDLECREATEDIAGRAM`

Parameter	Type	Description
`virtuality`	`integer`	Kind of virtuality. Only applicable to typed diagrams. Other diagrams should use the value `NOVIRTUAL` Possible values are: `NOVIRTUAL` `VIRTUAL` `REDEFINED` `FINALIZED` These values are defined in `sdtsym.h`
`DiagramType`	`integer`	Diagram type. See `sdtsym.h` for valid diagram types.
`name`	`string`	Diagram name.
`qualifier`	`string`	SDL qualifier. Could be empty.
`pageType`	`integer`	Type of the first page in the diagram. See `sdtsym.h` for valid page types. Note that the `pageType` must correspond with the `diagramType`.
`pageName`	`string`	Name of the first page in the diagram.

Service Reply

SESDLECREATEDIAGRAMREPLY

Parameter	Type	Description
status	integer	Service reply status.
bufId	integer	Refers to a buffer in the editor.

Errors

SDLE is busy, syntax error in text

Emitted Notifications

Notification	Description
SESDLECREATENOTIFY	As a result of creating the diagram.
SEPAGENOTIFY	As a result of creating a page.

Create MSC Diagram

Description

Creates an empty MSC diagram in new buffer. Corresponds to MSC Editor command *New*.

Tools Supporting the Service

SET_MSCE

Service Request

SEMSCECREATEDIAGRAM

Parameter	Type	Description
diagramType	integer	Diagram type. Defined in sdtsym.h. The value MSCDIAGRAM (16) should be used.
name	string	Name of the diagram to create.
qualifier	string	For future use. Should be empty.

Service Reply

SEMSCECREATEDIAGRAMREPLY

Parameter	Type	Description
status	integer	Service reply status.
bufId	integer	Refers to a buffer in the editor.

Errors

-

Emitted Notifications

Notification	Description
SEMSCENEWNOTIFY	As a result of creating the diagram.

Create OM Diagram

Description

Creates an empty OM diagram in new buffer. Corresponds to OM Editor command *New*.

Tools Supporting the Service

SET_OME

Service Request

SEOMECREATEDIAGRAM

Parameter	Type	Description
diagramType	integer	Diagram type. Defined in sdtsym.h. The value CLASSDIAGRAM (21) should be used.
name	string	Name of the diagram to create.
pageType	integer	Page type Defined in sdtsym.h. The value CLASSPAGE (16) should be used
pageName	string	name of the first page

Service Reply

SEOMECREATEDIAGRAMREPLY

Parameter	Type	Description
status	integer	Service reply status.
bufId	integer	Refers to a buffer in the editor.

Errors

-

Emitted Notifications

Notification	Description
SEOMENEWNOTIFY	As a result of creating the diagram.

Create Text Diagram

Description

Creates an empty text diagram in new buffer. Corresponds to Text Editor command *New*.

Tools Supporting the Service

SET_TE

Service Request

SETECREATEDIAGRAM

Parameter	Type	Description
diagramType	integer	Diagram type. See sdtsym.h for valid diagram types.
name	string	Name of the diagram to create.

Service Reply

SETECREATEREPLY

Parameter	Type	Description
status	integer	Service reply status.
bufId	integer	Refers to a buffer in the editor.

Errors

-

Emitted Notifications

Notification	Description
SETENEWNOTIFY	As a result of creating the diagram.

Editor – Object Services

Select Object

Description

The *Select Object* service will highlight an object in a diagram. Note that this command does not show the buffer and the selected object in a window. The difference between this service and the *Obtain GR Reference* service is that the *Select Object* service does not require the diagram to be saved on a file.

Tools Supporting the Service

```
SET_SDLE
SET_MSCE
SET_OME
```

Service Request

```
SESELECTOBJECT
```

Parameter	Type	Description
bufId	integer	Buffer referencing the diagram in which an object should be shown.
objectid	integer	Identifier to the object to show.
row	integer	The row where to put the text cursor.
column	integer	The column where to put the text cursor in the row.
keepSelections	bool	Flag indicating if old selections should be kept.

Service Reply

```
SESELECTOBJECTREPLY
```

Parameter	Type	Description
status	integer	Service reply status.

Errors

```
Invalid diagram buffer id
Invalid object id
SDLE is busy, syntax error in text    (SDLE only)
```

Show Object

Description

The *Show Object* service will make sure that the specified object is visible in a window. This means that it will display the buffer in a window and if necessary scroll into the position where the object is. Often used in combination with *Select Object*.

Tools Supporting the Service

```
SET_SDLE
SET_MSCE
SET_OME
```

Service Request

```
SESHOWOBJECT
```

Parameter	Type	Description
bufId	integer	Buffer referencing the diagram in which an object should be shown.
objectId	integer	Identifier to the object to show.

Service Reply

```
SESHOWOBJECTREPLY
```

Parameter	Type	Description
status	integer	Service reply status.

Errors

```
Invalid diagram buffer id
Invalid object id
SDLE is busy, syntax error in text    (SDLE only)
```

Insert SDL Object

Description

Adds an object to the SDL diagram.

Tools Supporting the Service

SET_SDLE

Service Request

SESDLEINSERTOBJECT

Parameter	Type	Description
bufId	integer	Buffer in which to insert the object.
pageName	string	Page to insert the object on.
shiftIsDown	bool	Emulate behavior of having <Shift> pressed when inserting an symbol.
objectType	integer	Type of object. Valid objects depend on the type of diagram and if syntax checking is on. Corresponds to available symbols in the editor symbol menu. **NOTE: SDL reference symbols are not possible to add using this service**. Definitions of symbols are found in sdtsym.h.
objectText	string	Text in object.

Service Reply

SESDLEINSERTOBJECTREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

```
Invalid diagram buffer id
Invalid object id
A reference symbol containing text\
cannot be inserted in diagram <diagram>
Object type <objectType> is not allowed\
in diagram <diagram>
Unable to open page
The page is too small to insert the object
SDLE is busy, syntax error in text
```

Insert MSC Object

Description

The *Insert Object* service will create a new object in the diagram identified by the parameter and return an object identification (an integer) to the client. It will not display the new object.

For a full specification of how to specify the object to insert, the specification of Message Sequence Charts Z.120 should be consulted or the document “MSC Trace and Log Format 2.0 Specification” available from IBM Rational.

Tools Supporting the Service

`SET_MSCE`

Service Request

`SEINSERTOBJECT`

Parameter	Type	Description
`bufId`	`integer`	Buffer in which to insert the object.
`afterObject`	`integer`	Only object id 0 is allowed, and adds the object as the last object.
`objectDescription`	`string`	Description of the object to insert. The description should be in accordance with Z.120 using *event oriented* PR. Only one object at the time can be inserted.

Service Reply

`SEINSERTOBJECTREPLY`

Parameter	Type	Description
`status`	`integer`	Service reply status.
`objectId`	`integer`	Identifier to object.

Errors

`Invalid diagram buffer id`
`Errors from parsing objectDescription`

Remove Object

Description

The *Remove Object* service will delete an object in a diagram.

Tools Supporting the Service

SET_MSCE

Service Request

SEREMOVEOBJECT

Parameter	Type	Description
bufId	integer	Buffer referencing the diagram in which an object should be removed.
objectId	integer	The object to remove.

Service Reply

SEREMOVEOBJECTREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

```
Invalid diagram buffer id
Invalid object id
Object can not be removed   (MSCE only)
```

Get Object Text

Description

The service will extract all texts for a given object.

Tools Supporting the Service

```
SET_SDLE
SET_MSCE
SET_OME
```

Service Request

```
SEGETOBJECTTEXT
```

Parameter	Type	Description
`bufid`	`string`	A buffer referencing the diagram where the object exists.
`objectId`	`integer`	Identifier to the object.

Service Reply

```
SEGETOBJECTTEXTREPLY
```

Parameter	Type	Description
`status`	`integer`	Service reply status.
`objectType`	`integer`	The type of the object. Definitions of possible object types are found in `sdtsym.h`.
`textnumber`	`integer`	The number of the text strings.
`texts`	`stringlist`	All the texts associated to the object.

Errors

```
Invalid diagram buffer id
No page found with this object id
No object with this id found
```

Editor – Object Attribute Services

Introduction

It is possible to extend the data associated to an object managed by the SDL Suite with the user's own data. How this is done is described in Figure 161.

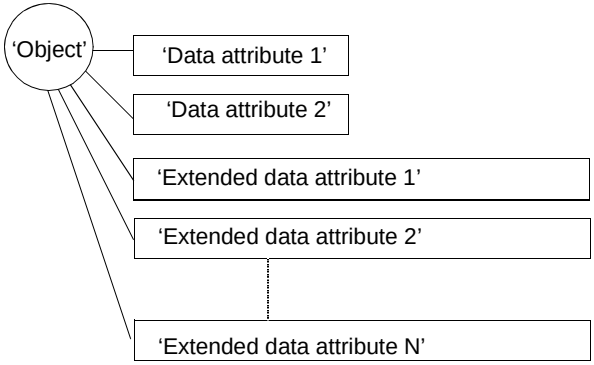


Figure 161: Extending the attributes associated to a symbol

The object can have any number of attributes associated to it, not necessarily 2 as illustrated. The number of extended data attributes is arbitrary.

An object is potentially any item that is handled as a source component by the SDL Suite, or any of the sub-components of which it consists. Objects are thus:

- SDL diagram
- MSC diagram
- OM, SC, HMSC diagram
- SDL Page
- OM, SC, HMSC page
- SDL symbol
- MSC symbol
- OM, SC, HMSC symbol

The purpose of the *extended data attribute* is:

- To allow the user to add his own-defined data to diagrams managed by the SDL Suite.

- In addition, data extensions can by connected to operations to be performed on that data. Operations are performed through PostMaster messages or Operating System commands.

Extended Data Attribute

An extended data attribute defines a number of fields. Below is a few of them further elaborated.

- `ProprietaryKey`

  Provides a means to tag extensions with the originating company / organization. Should normally be set to 0.

- `AttributeKey`

  Up to the user's preference when deciding how to use this key.

- `data_length`

  Provides information about how many bytes of data the `variant_data` union consists of. Generally, all elements of variable length need to have their length specified, since the user must be able to store hex 0 as data.

- `data_interpretation` and `data`

  In the SDL Suite, data is always interpreted as *raw data*. The SDL Suite have no knowledge about the format and meaning of the data. *Raw data* may consist of information produced by an external tool. Its format is not known by the SDL Suite. Thus, operation of any sort applied on that data, cannot be performed.

- `comment_length`

  How many bytes the comment consists of.

- `comment`

  Gives the possibility to attach a (preferable) readable comment in plain text. The comment could assist the user in understanding the meaning of data that he has attached to the object.

Display Key

Description

Request to display, i.e. select, all symbols referred by `SDTRef`, that have at least one associated attribute that matches the keys `ProprietaryKey` and `AttributeKey`.

- `AttributeKey` will be omitted if it has value 0
- If `ProprietaryKey` is 0 both keys will be omitted.

`SDTRef` is typically a diagram, but could also be a page or a specific symbol. The symbols that match the criteria will be marked as selected in the drawing area. Other symbols will be de-selected.

- If `SDTRef` is empty, display all symbols is all diagrams that are read by the tool, and that match the keys
- If `SDTRef` denotes a diagram, display all symbols in the diagram that match the keys
- If `SDTRef` denotes a page, display all symbols in the page that match the keys
- If `SDTRef` denotes a symbol, display the symbol if it matches the keys.

Tools Supporting the Service

SET_SDLE
SET_MSCE
SET_OME

Service Request

SEDISPLAYKEY

Parameter	Type	Description
ProprietaryKey	integer	See the introductory description.
AttributeKey	integer	See the introductory description.
SDTRef	string	See the introductory description.

Service Reply

SEDISPLAYKEYREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

SDT Reference Errors

List Key

Description

Request to list all objects referred by SDTRef, that have at least one associated attribute that matches the keys ProprietaryKey and AttributeKey.

- AttributeKey will be omitted if it has value 0.
- If ProprietaryKey is 0 both keys will be omitted.

SDTRef is typically a diagram, but could also be a page or a specific symbol. The response is a count and a list of SDTRef to all objects that match the keys.

- If SDTRef is NULL, list all symbols is all diagrams that are read by the tool, and that match the keys.
- If SDTRef denotes a diagram, return SDTRef to all objects in the diagram that match the keys.
- If SDTRef denotes a page, return SDTRef to objects in the page that match the keys.
- If SDTRef denotes an object, return the SDTRef to the object if it matches the keys.

Tools Supporting the Service

SET_SDLE
SET_MSCE
SET_OME

Service Request

SELISTKEY

Parameter	Type	Description
ProprietaryKey	integer	See the introductory description.
AttributeKey	integer	See the introductory description.
SDTRef	string	See the introductory description.

Service Reply

SELISTKEYREPLY

Parameter	Type	Description
status	integer	Service reply status.
NoOfKeys	integer	Number of matching keys.
SDTRef*	string	A list of SDT references.

Errors

SDT Reference Errors

Create Attribute

Description

Request add a new attribute to the symbol that matches the SDT reference `SDTRef`.

Tools Supporting the Service

`SET_SDLE`
`SET_MSCE`
`SET_OME`

Service Request

`SECREATEATTRIBUTE`

Parameter	Type	Description
`ProprietaryKey`	`integer`	A key that allows the external user / tool to distinguish his extensions from other users' / tools'.
`AttributeKey`	`integer`	A key that allows the external user / tool to classify his own extension.
`SDTRef`	`string`	A valid SDT reference identifying the object.
`datainterpret`	`integer`	Defines how `data` is to be interpreted. Should be set to 0.
`comment`	`string`	An explanatory (readable) comment.
`MenuChoice`	`string`	The name of the pop-up menu choice that will be appended to the tool's pop-up menu, upon selection of the symbol.
`dataLen`	`integer`	Length of data.
`data`	`ByteString`	The user's data.

Service Reply

`SECREATEATTRIBUTEREPLY`

Parameter	Type	Description
status	integer	Service reply status.

Errors

SDT Reference Errors
Error in match

Update Attribute

Description

Request to update the attribute that matches the search criteria (`SDTRef`, `ProprietaryKey`, `AttributeKey`). If an attribute matches the search keys, then the entire attribute's contents will be updated (replaced) with new contents.

Tools Supporting the Service

```
SET_SDLE
SET_MSCE
SET_OME
```

Service Request

```
SEUPDATEATTRIBUTE
```

Parameter	Type	Description
`ProprietaryKey`	`integer`	A key that allows the external user / tool to distinguish his extensions from other users' / tools'.
`AttributeKey`	`integer`	A key that allows the external user / tool to classify his own extension.
`SDTRef`	`string`	A valid SDT reference identifying the object.
`datainterpret`	`integer`	Defines how `data` is to be interpreted. The following values are recognized: Should be set to 0.
`comment`	`string`	An explanatory (readable) comment.
`MenuChoice`	`string`	The name of the pop-up menu choice that will be appended to the tool's pop-up menu, upon selection of the symbol.
`dataLen`	`integer`	Length of data.

Parameter	Type	Description
data	ByteString	The user's data.

Service Reply

SEUPDATEATTRIBUTEREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

SDT Reference Errors
Error in match

Read Attribute

Description

Request to read the attribute(s) that match(es) the keys and the diagram identified by `SDTRef`. The `SDTRef` reference should contain a page attribute.

Tools Supporting the Service

SET_SDLE
SET_MSCE
SET_OME

Service Request

SEREADATTRIBUTE

Parameter	Type	Description
ProprietaryKey	integer	See the introductory description.
AttributeKey	integer	See the introductory description.
SDTRef	string	See the introductory description.

Service Reply

SEREADATTRIBUTEREPLY

Parameter	Type	Description
status	integer	Service reply status.
DataInterpret	integer	Type of data, always 0.
comment	string	Comment text.
MenuChoice	string	The defined menu choice.
dataLen	integer	Length of binary data.
data	ByteString	Data associated to the extended attribute.

Errors

SDT Reference Errors
Error in match

Delete Attribute

Description

Request to delete the attribute that matches the search criteria (`SDTRef`, `ProprietaryKey`, `AttributeKey`).

Tools Supporting the Service

```
SET_SDLE
SET_MSCE
SET_OME
```

Service Request

```
SEDELETEATTRIBUTE
```

Parameter	Type	Description
`ProprietaryKey`	`integer`	See the introductory description.
`AttributeKey`	`integer`	See the introductory description.
`SDTRef`	`string`	See the introductory description.

Service Reply

```
SEDELETEATTRIBUTEREPLY
```

Parameter	Type	Description
`status`	`integer`	Service reply status.

Errors

SDT Reference Errors

```
More than one object match the search
No object match the search
```

Information Server Services

Load Definition File

Description

Loads a file containing external signal definitions into the Information Server. The contents of such a file are then made available via the *Signal Dictionary* functionality found in the SDL Editor.

The Information Server could read any ASCII file, but for efficient usage, the format of the files should contain one signal definition per line.

Tools Supporting the Service

`SET_INFOSERVER`

Service Request

`SELOADDEFINITIONMAP`

Parameter	Type	Description
`fileName`	`string`	Name of the file to load.
`tag`	`integer`	Is used to either add or remove a file. 0 = Add definition 1 = Remove definition.

Service Reply

`SELOADDEFINITIONMAPREPLY`

Parameter	Type	Description
`status`	`integer`	Service reply status.

Errors

```
File already added
File was not added before
```

SDL Editor Services

GRPR

Description

Generate PR for a given binary file. Optionally, GR references and CIF comments could be generated.

If the file is already being edited the PR generated will be from the internal version in the SDL Editor and not from the data in the file.

If the diagram is not being edited it will be unloaded after the PR generation.

Tools Supporting the Service

`SET_SDLE`

Service Request

`SEGRPRP`

Parameter	Type	Description
`fileName`	`string`	The file where the SDL/GR is stored.
`fileName`	`string`	The file where the PR is written
`append`	`bool`	If *true*, the PR will be appended to the file.
`generateGRRef`	`bool`	If *true*, the generated PR will also contain graphical references, that are used in the Analyzer to backtrace errors.
`generateCIF`	`bool`	If *true*, the PR will also contain CIF comments.

Service Reply

`SEGRPRREPLY`

Parameter	Type	Description
`status`	`integer`	Service reply status.

Errors

```
Invalid file name
SDLE is busy, syntax error in text    (SDLE only)
```

Tidy Up

Description

The *Tidy Up* service will perform a tidy up on a specified diagram. The functionality is identical to the *Tidy Up* command available in the SDL Editor.

Tools Supporting the Service

SET_SDLE

Service Request

SETIDYUP

Parameter	Type	Description
bufid	integer	Buffer referencing the diagram to tidy up.

Service Reply

SETIDYUPREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

Invalid diagram buffer id

SC Editor Services

Get Diagram Info

Description

Get information about the symbols and lines of a SC diagram. The format of the returned *infoList* is:

```
Symbols: <noOfSymbols>
<type> <id> <superstateId> <noOfTexts> <text1> ...
...
Lines: <noOfLines>
<type> <fromSymbolId> <toSymbolId> <noOfTexts> <text1> ...
...


Where:
<type> ::=      An integer denoting the type of symbol or line
                Types can be found in sdtsym.h
<id> ::=        An integer identifying a symbol
```

Tools Supporting the Service

SET_OME

Service Request

SEGETDIAGRAMINFO

Parameter	Type	Description
bufId	integer	Refers to a buffer in the editor.

Service Reply

SEGETDIAGRAMINFOREPLY

Parameter	Type	Description
status	integer	Service reply status.
infoList	stringList	The symbols and lines of the diagram.

Errors

Invalid diagram buffer id
Invalid diagram type

MSC Editor Services

MSC GRPR

Description

Generate PR for a given MSC binary file. Optionally, GR references can be generated.

If the file is already being edited, the generated PR will be from the internal version in the MSC Editor and not from the data in the file.

If the diagram is not being edited it will be unloaded after the PR generation.

Tools Supporting the Service

SET_MSCE

Service Request

SEMSCGRPR

Parameter	Type	Description
fileName	string	The file where the MSC/GR is stored.
fileName	string	The file where the PR is written.
mscDocName	string	The name of the MSC document. If empty the PR will be appended to PR file.
generateGRRef	bool	If *true*, the generated PR will also contain graphical references, that are used in the Analyzer to backtrace errors.

Service Reply

SEMSCGRPRREPLY

Parameter	Type	Description
errors	integer	Number of errors in the GR file.
warnings	integer	Number of warnings in the GR file.
errorLog	string	Description of errors and warnings, with GR references.

MSC Generate cui script

Description

Generate cui script for a given MSC diagram.

If the file is already being edited, the generated script will be from the internal version in the MSC Editor and not from the data in the file.

If the diagram is not being edited it will be unloaded after the script generation.

The input script will be generated as a .cui file with the same name as the input MSC diagram. The .cui file will be located in the same directory as your input MSC.

Tools Supporting the Service

`SET_MSCE`

Service Request

`SEGENERATEINPUTSCRIPT`

Parameter	Type	Description
`fileName`	`string`	The file where the MSC/GR is stored.

Service Reply

`SEGENERATEINPUTSCRIPTREPLY`

Parameter	Type	Description
`status`	`integer`	Service reply status.

HMSC Editor Services

HMSC GRPR

Description

Generate PR for a given HMSC binary file. Optionally, GR references can be generated.

If the file is already being edited, the generated PR will be from the internal version in the HMSC Editor and not from the data in the file.

If the diagram is not being edited it will be unloaded after the PR generation.

Tools Supporting the Service

SET_OME

Service Request

SEHMSCGRPR

Parameter	Type	Description
fileName	string	The file where the HMSC/GR is stored.
fileName	string	The file where the PR is written.
mscDocName	string	The name of the HMSC document. If empty the PR will be appended to PR file.
generateGRRef	bool	If *true*, the generated PR will also contain graphical references, that are used in the Analyzer to backtrace errors.

Service Reply

SEHMSCGRPRREPLY

Parameter	Type	Description
errors	integer	Number of errors in the GR file.
warnings	integer	Number of warnings in the GR file.
errorLog	string	Description of errors and warnings, with GR references.

CIF Services

The CIF services enables the user to create diagrams where the graphical objects and texts could be positioned in a controlled manner.

These services enables the user to build converters to SDL Suite diagram formats.

The diagram types which are supported are:

- Block interaction diagrams,
- Process interaction diagrams

where the services are supported by the SDL Editor, and

- OM diagrams,

where the services are supported by the OM Editor.

Create SDL Diagram

Description

Creates an empty SDL diagram in a new buffer. Corresponds to the SDL Editor command *New*. The defaults applied to the created page are defined for the *Create SDL Page* service.

The texts for the package reference and the headings will be shown on every page in the diagram. The package reference symbol and the heading symbols will be automatically created, but their size can be changed by using the *Insert SDL Object* service for these symbols, (see "Object Specific Parameters" on page 641).

The kernel heading will be parsed to extract the necessary attributes like the virtuality, diagram type and diagram name. If this parsing fails the service will return an error.

The splitting of the SDL heading into a kernel heading and an additional heading is done according to the rules in section 2.2.5 'Partitioning of diagrams' in Z.100 with some necessary exceptions. Normally the kernel heading contains the heading up to, and including the diagram name,

```
<kernel heading> ::= [<virtuality>] <diagram kind>
{<diagram name> | <diagram identifier>}
```

The following exceptions to this rule apply:

For a type based system diagram the kernel heading contains the complete definition except for the ending <end> clause.

```
<kernel heading> ::= <typebased system heading>
```

For a process diagram the <number of process instance> might be included in the kernel heading.

For a procedure diagram the kernel heading is

```
<kernel heading> ::= <procedure preamble> procedure
{<procedure name> | <procedure identifier>}
```

Tools Supporting the Service

SET_SDLE

Service Request

SESDLECIFCREATEDIAGRAM

Parameter	Type	Description
packageReferenceText	string	The text that will appear in the package reference symbol
kernelHeadingText	string	The text that will appear in the kernel heading symbol
AdditionalHeadingText	string	The text that will appear in the additional heading symbol
pageType	integer	Type of the first page in the diagram. See sdtsym.h for valid page types. Note that the pageType must correspond with the diagramType.
pageName	string	Name of the first page in the diagram.
pageWidth	integer	The width of the page.
pageHeight	integer	The height of the page.
gridWidth	integer	The grid width.

Parameter	Type	Description
gridHeight	integer	The grid height.
autonumbered	bool	If true the page is an auto-numbered page.
showMeFirst	bool	If true this page is set as the ShowMeFirst page.
scale	integer	The scale used when shown, 100 means a 1:1 scale.

Service Reply

SESDLECIFCREATEDIAGRAMREPLY

Parameter	Type	Description
status	integer	Service reply status.
bufid	integer	Refers to a buffer in the editor.

Errors

```
The pagename is empty
Illegal diagram kernel heading
```

Emitted Notifications

Notification	Description
SESDLENEWNOTIFY	As a result of creating the diagram.
SEPAGENOTIFY	As a result of creating a page.

Create SDL Page

Description

The *Create Page* service will create a new page in an existing diagram. The new page will be inserted as the last page in the diagram. The created page will have a frame symbol with margins to the page border as defined by the actual preference file. The kernel heading, additional heading and page number symbols exists and are placed automatically relative to the frame symbol. The package reference symbol exists, automatically placed, if the page type is a system page or a package page. The frame symbol can be changed to another position by using the Insert SDL Object service.

If a page already exists with the given page name, the service will be ignored and the return status is OK. This can be used for convenience to always send Create Page for all pages including the one already sent in the *Create SDL Diagram*.

The grid values should be multiples of 5 mm. If the values are zero or less than zero this means that the preferences values are used for the grid values.

A page can be set to be autonumbered. When this is used a page name must be given as well. For consecutive autonumbered pages these names must be “1”, “2”, “3” and so on. It will be checked that the given page name is the same as the generated page name. If not an error will be returned.

If the *ShowMeFirst* attribute is set for a new page it will override an already existing ShowMeFirst page.

Tools Supporting the Service

`SET_SDLE`

Service Request

`SESDLECIFCREATEPAGE`

Parameter	Type	Description
`bufId`	`integer`	Buffer referencing the diagram in which the page will be created.

Parameter	Type	Description
pageType	integer	Type of the page in the diagram. See sdtsym.h for valid page types. Note that the pageType must correspond with the diagramType.
pageName	string	Name of the new page in the diagram.
pageWidth	integer	The width of the page.
pageHeight	integer	The height of the page.
gridWidth	integer	The grid width.
gridHeight	integer	The grid height.
autonumbered	bool	If true the page is an autonumbered page.
showMeFirst	bool	If true this page is set as the ShowMe-First page.
scale	integer	The scale used when shown, 100 means a 1:1 scale.

Service Reply

SESDLECIFCREATEPAGEREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

```
Invalid diagram buffer id
The pagename is empty
Command canceled in add page
For autonumbered pages, the given pagename "2" must
be the same as the generated page name "1"
```

Emitted Notifications

Notification	Description
SEPAGENOTIFY	As a result of creating a page.

Insert SDL Object

Description

The *Insert Object* service will insert an object in a specified position at a specific page. The return value is the object id for the inserted object. Lines connected to the inserted symbol should refer to the *objectId* returned by this service.

The following checks are applied on parameters:

- If a line is inserted the symbols connecting the line must already have been inserted.
- For lines it is an error if there are less than two breakpoints.
- The object type must be a known symbol or line.

The following is a non-exhaustive list of restrictions, applied inside SDLE when manipulating with symbols, that are **not** checked by this service:

- Ratio for process symbols, should be 1:2 (height vs. width).
- Positions outside the page area.
- Text syntax for reference symbols.
- Illegal texts on flowlines.
- Sizes less than minimum allowed.
- That lines are connected to symbols with the correct syntax.
- That line endpoints fall on enclosing rectangle of the symbol it is connected to.
- Symbols placed on pages where they should not appear, e.g. a task on a system page.
- Overlapping symbols

Tools Supporting the Service

`SET_SDLE`

Service Request

SESDLECIFINSERTOBJECT

Parameter	Type	Description
bufId	integer	Buffer referencing the diagram in which an object should be shown.
pagename	string	An existing page within the diagram.
objectType	integer	Identify the type of object to insert. Valid values for all symbols are given in sdtsym.h. For lines the values are listed below.
updateFlag	bool	If this flag is true a routine will be called for this page to fix some attributes for the objects. This fixing must be done before the page is saved. If this flag is false this fix will not be done.
object specific parameters	varying	One or more parameters defining the inserted object.

Service Reply

SESDLECIFINSERTOBJECTREPLY

Parameter	Type	Description
status	integer	Service reply status.
objectId	integer	Identifier to the inserted object.

Errors

```
Invalid diagram buffer id
Invalid pagename
Invalid fromSymbol id for a line:
Invalid toSymbol id for a line:
Invalid fromSymbol. The fromSymbol is a line:
Invalid toSymbol. The toSymbol is a line:
Number of points in line < 2 :
Invalid objectId for ConnectionPoint line:
Invalid object type:
```

Object Specific Parameters

For each object type the needed parameters are listed. All coordinates are given in 1/10 mm units in the coordinate system with the upper left corner of the page as origo and the y-coordinate having the positive direction going downwards.

A given coordinate will be adjusted to fall on a grid. For symbols this grid is given by a grid net of 5 x 5 mm. For line breakpoints and text positions the grid net is 2.5 x 2.5 mm.

The *Pagenumber* symbol (objectType number 53) has no extra specific parameters. When inserted there will not be a new symbol created as the pagenumber symbol is always created when the page is created. The text for the pagenumber symbol is generated automatically. The only use of inserting this symbol is to force an update of the page by having the *updateFlag* set to *true*.

- **Process Interaction diagram symbols**

```
Start(1), ProcedureStart(2), MacroInlet(3), Stop(4),
ProcedureReturn(5), MacroOutlet(6), State(7),
InputRight(8), InputLeft(9), PriorityInputRight(10),
PriorityInputLeft(11), ContinuousSignal(12),
EnablingCondition(13), Task(14), OutputRight(15),
OutputLeft(16), PriorityOutputRight(17),
PriorityOutputLeft(18), ProcedureCall(19),
InConnector(20), CreateRequest(21), Decision(22),
TransitionOption(23), MacroCall(24),
OutConnector(25), Save(26)
```

Parameter	Type	Description
`left`	`integer`	The x coordinate for the left side.
`top`	`integer`	The y coordinate for the top side.
`width`	`integer`	The width of the symbol.
`height`	`integer`	The height of the symbol.
`text`	`string`	The text in the symbol. The text is automatically aligned according to the built-in routines in SDLE

- **Block Interaction diagram symbols**

```
Procedure(27), Service(41), Process(42), Block(43),
BlockSubstructure(45), ChannelSubstructure(46),
SystemType(47), BlockType(48), ServiceType(49),
ProcessType(50), Operator(56).
```

Parameter	Type	Description
left	integer	The x coordinate for the left side.
top	integer	The y coordinate for the top side.
width	integer	The width of the symbol.
height	integer	The height of the symbol.
dashed	bool	If true the symbol will be dashed, this is valid for block, process and service symbols.
text	string	The text in the symbol.
XtextPosition	integer	The x coordinate for the left top position of the text.
YtextPosition	integer	The y coordinate for the left top position of the text.

There is no check that the text will fall inside the symbol boundary.

- **TextSymbol**(28)

Parameter	Type	Description
left	integer	The x coordinate for the left side.
top	integer	The y coordinate for the top side.
width	integer	The width of the symbol.
height	integer	The height of the symbol.
text	sString	The text in the symbol.

An automatically sized *TextSymbol* shall have the *width* and *height* set to zero. If values are given this means that the symbol will be clipped using the given sizing values.

- **Comment**(29), **CommentLeft**(30), **TextExtension**(31) and **TextExtensionLeft**(32)

Parameter	Type	Description
`left`	`integer`	The x coordinate for the left side (Comment, TextExtension) or the right side (CommentLeft, TextExtensionLeft)
`top`	`integer`	The y coordinate for the top side.
`text`	`string`	The text in the symbol.

For a *CommentLeft* or *TextExtensionLeft* the position gives the right top corner of the symbol. The size is automatically adjusted.

- **FrameSymbol**(44)

Parameter	Type	Description
`left`	`integer`	The x coordinate for the left side.
`top`	`integer`	The y coordinate for the top side.
`width`	`integer`	The width of the symbol.
`height`	`integer`	The height of the symbol.

A default *FrameSymbol* will always exist on a created page. This service will replace the existing Frame with a new frame symbol at the given position with the new size. The existing *heading symbols*, *package reference symbol* and the *pagenumber symbol* will be automatically adjusted.

- **AdditionalHeading**(52), **PackageReference**(55)

Parameter	Type	Description
`width`	`integer`	The width of the symbol.
`height`	`integer`	The height of the symbol.

For automatically sized symbols the values for the *width* and the *height* should be zero. If values different from zero are given this means that the symbol will be clipped.

Note that it might be necessary to insert this symbol for every page. This is so because these symbols might have a unique size for every page. If the symbols are not clipped there is no need to insert the symbol as they will be automatically generated as an unclipped symbol.

The size of the kernel heading cannot be changed.

- **FlowLine**(101)

Parameter	Type	Description
fromID	integer	The objectId for the symbol where the line starts.
toID	integer	The objectId for the symbol where the line ends.
textExists	bool	Is *true* if there is text on the flow line.
text	string	Optional text in the message. Only exists if textExists is true.
XtextPosition	integer	Optional text position in the message. Only exists if textExists is true. x coordinate for the left top corner of the text.
YtextPosition	integer	Optional text position in the message. Only exists if textExists is true. y coordinate for the left top corner of the text.
dashed	bool	If *true* the line will be dashed.
numPoints	integer	The number of breakpoints for the line. There must be at least 2.
x	integer	x coordinate for a point.
y	integer	y coordinate for a point.

It is checked that symbols exist having the given *fromID* and *toID*. The text and text position shall only exist if the *textExists* flag is *true*. *Flowlines* after *Decision*, *TransitionOption* and *MacroCall*

should always have text. There will be *numPoints x* and *y*-values at the end of the message. Line coordinates are adjusted to the nearest grid point but it will not be checked whether line endpoints are placed on the symbol boundaries, as they are expected to be.

- **SignalRoute**(102)

Parameter	Type	Description
fromID	integer	The objectId for the symbol where the line starts. Zero means ENV.
toID	integer	The objectId for the symbol where the line ends. Zero means ENV.
name	string	The name of the signal route.
signalList	string	The text in the signal list.
XnamePosition	integer	The x coordinate for the left top position of the name.
YnamePosition	integer	The y coordinate for the left top position of the name.
XsignalListPosition	integer	The x coordinate for the left top position of the signal list.
YsignalListPosition	integer	The y coordinate for the left top position of the signal list.
numPoints	integer	The number of breakpoints for the line. There must be at least 2.
x	integer	x coordinate for a point.
y	integer	y coordinate for a point.

It is checked that symbols exists having the given *fromID* and *toID*. There will be *numPoints x* and *y*-values at the end of the message. Text position and line coordinates are adjusted to the nearest grid point but it will not be checked whether line endpoints are placed on the symbol boundary (the rectangle enclosing the symbol), as they are expected to be. Extra line segments needed for e.g. a signal route entering a service will be automatically generated.

- **DoubleSignalRoute**(103)

Parameter	Type	Description
`fromID`	`integer`	The objectId for the symbol where the line starts. Zero means ENV.
`toID`	`integer`	The objectId for the symbol where the line ends. Zero means ENV.
`name`	`string`	The name of the signal route.
`signalList1`	`string`	The text in the signal list in the direction from fromID to toID.
`signalList2`	`string`	The text in the signal list in the reversed direction.
`XnamePosition`	`integer`	The x coordinate for the left top position of the name.
`YnamePosition`	`integer`	The y coordinate for the left top position of the name.
`XsignalList1Position`	`integer`	The x coordinate for the left top position of the first signal list.
`YsignalList1Position`	`integer`	The y coordinate for the left top position of the first signal list.

Parameter	Type	Description
XsignalList2Position	integer	The x coordinate for the left top position of the second signal list.
YsignalList2Position	integer	The y coordinate for the left top position of the second signal list.
numPoints	integer	The number of break-points for the line. There must be at least 2.
x	integer	x coordinate for a point.
y	integer	y coordinate for a point.

- **Channel**(104)

Parameter	Type	Description
fromID	integer	The objectId for the symbol where the line starts. Zero means ENV.
toID	integer	The objectId for the symbol where the line ends. Zero means ENV.
name	string	The name of the channel.
signalList	string	The text in the signal list.
XnamePosition	integer	The x coordinate for the left top position of the name.
YnamePosition	integer	The y coordinate for the left top position of the name.
XsignalListPosition	integer	The x coordinate for the left top position of the signal list.

Parameter	Type	Description
YsignalListPosition	integer	The y coordinate for the left top position of the signal list.
XarrowPosition	integer	The x coordinate for the arrow.
YarrowPosition	integer	The y coordinate for the arrow.
noDelay	bool	If true the channel is a non delaying channel.
numPoints	integer	The number of breakpoints for the line. There must be at least 2.
x	integer	x coordinate for a point.
y	integer	y coordinate for a point.

It is checked that symbols exists having the given *fromID* and *toID*. There will be *numPoints x* and *y*-values at the end of the message. Text position and line coordinates are adjusted to the nearest grid point but it will not be checked whether line endpoints are placed on the symbol boundary (the rectangle enclosing the symbol), as they are expected to be. The arrow position are automatically adjusted to be placed at the line as close as possible to the given arrow point. The angle of the arrow is automatically calculated. For a non delaying channel the arrow position is ignored and it will be automatically placed at the last breakpoint.

- **DoubleChannel**(104)

Parameter	Type	Description
fromID	integer	The objectId for the symbol where the line starts. Zero means ENV.
toID	integer	The objectId for the symbol where the line ends. Zero means ENV.
name	string	The name of the channel.
signalList1	string	The text in the signal list in the direction from fromID to toID.
signalList2	sString	The text in the signal list in the reversed direction.
XnamePosition	integer	The x coordinate for the left top position of the name.
YnamePosition	integer	The y coordinate for the left top position of the name.
XsignalList1Position	integer	The x coordinate for the left top position of the first signal list.
YsignalList1Position	integer	The y coordinate for the left top position of the first signal list.
XsignalList2Position	integer	The x coordinate for the left top position of the second signal list.
YsignalList2Position	integer	The y coordinate for the left top position of the second signal list.
Xarrow1Position	integer	The x coordinate for the first arrow.

Parameter	Type	Description
Yarrow1Position	integer	The y coordinate for the first arrow.
Xarrow2Position	integer	The x coordinate for the second arrow.
Yarrow2Position	integer	The y coordinate for the second arrow.
noDelay	bool	If true the channel is a non delaying channel.
numPoints	integer	The number of breakpoints for the line. There must be at least 2.
x	integer	x coordinate for a point.
y	integer	y coordinate for a point.

- **CreateLine**(106), **ChannelSubstructureLine**(107)

Parameter	Type	Description
fromID	integer	The objectId for the symbol where the line starts. This object is a channel for the ChannelSubstructureLine.
toID	integer	The objectId for the symbol where the line ends.
numPoints	integer	The number of breakpoints for the line. There must be at least 2.
x	integer	x coordinate for a point.
y	integer	y coordinate for a point.

It is checked that symbols exists having the given *fromID* and *toID*. There will be *numPoints x* and *y*-values at the end of the message. Line coordinates are adjusted to the nearest grid point but it will not be checked whether line endpoints are placed on the symbol boundary, as they are expected to be. For a *ChannelSubstructureLine* the starting is adjusted to be placed on the channel.

- **GateIn**(109), **GateOut**(110)

Parameter	Type	Description
X	integer	The x coordinate for the gate position on the FrameSymbol boundary.
Y	integer	The y coordinate for the gate position on the FrameSymbol boundary.
dashed	bool	If true the gate is dashed.
name	string	The name of the gate.
signalList	string	The text in the signal list.
constraint	string	The text in the constraint symbol connected to the gate.
XnamePosition	integer	The x coordinate for the left top position of the name.
YnamePosition	integer	The y coordinate for the left top position of the name.
XsignalListPosition	integer	The x coordinate for the left top position of the signal list.
YsignalListPosition	integer	The y coordinate for the left top position of the signal list.

The length of the gate is automatically calculated, so also the arrow position. The point will be adjusted such that it is guaranteed to stay on the Frame boundary. If the frame symbol is inserted after a gate, the gate might be positioned outside the frame. The correct positions will be fixed in the update that must be performed before saving the page.

- **DoubleGate**(111)

Parameter	Type	Description
X	integer	The x coordinate for the gate position on the FrameSymbol boundary.
Y	integer	The y coordinate for the gate position on the FrameSymbol boundary.
dashed	bool	If true the gate is dashed.
name	string	The name of the gate.
signalList1	string	The text in the signal list in the direction into the diagram.
signalList2	string	The text in the signal list in the direction out of the diagram.
constraint	string	The text in the constraint symbol connected to the gate.
XnamePosition	integer	The x coordinate for the left top position of the name.
YnamePosition	integer	The y coordinate for the left top position of the name.
XsignalList1Position	integer	The x coordinate for the left top position of the first signal list.
YsignalList1Position	integer	The y coordinate for the left top position of the first signal list.

Parameter	Type	Description
XsignalList2Position	integer	The x coordinate for the left top position of the second signal list.
YsignalList2Position	integer	The y coordinate for the left top position of the second signal list.

- **ConnectionPoint**(112)

Parameter	Type	Description
objectID	integer	The objectId for the symbol where the ConnectionPoint exists. Zero means ENV.
X	integer	The x coordinate for the ConnectionPoint.
Y	integer	The y coordinate for the ConnectionPoint.
name	string	The name of the ConnectionPoint.
XnamePosition	integer	The x coordinate for the left top position of the name.
YnamePosition	integer	The y coordinate for the left top position of the name.

A *ConnectionPoint* is used for gate names on the Frame symbol (ENV), channel names in block diagrams for graphical connect statements and for gate names inside block/process/service instances of block/process/service types. The *objectID* must be a valid symbol. The coordinate is adjusted such that the point will fall on the symbol boundary. If the frame symbol is inserted after a connection point residing on the frame, it might be positioned outside the frame. The correct positions will be fixed in the update that must be performed before saving the page.

Create OM Diagram

Description

Creates an empty OM diagram in a new buffer. Corresponds to the OM Editor command *New*. The defaults applied to the created page are defined for the Create OM Page service.

Tools Supporting the Service

SET_OME

Service Request

SEOMECIFCREATEDIAGRAM

Parameter	Type	Description
name	string	Diagram name.
pageName	string	Name of the first page in the diagram.
pageWidth	integer	The width of the page.
pageHeight	integer	The height of the page.
gridWidth	integer	The grid width.
gridHeight	integer	The grid height.
autonumbered	bool	If true the page is an autonumbered page.
showMeFirst	bool	If true this page is set as the ShowMeFirst page.
scale	integer	The scale used when shown, 100 means a 1:1 scale.

Service Reply

SEOMECIFCREATEDIAGRAMREPLY

Parameter	Type	Description
status	integer	Service reply status.
bufid	integer	Refers to a buffer in the editor

Errors

```
The pagename is empty
```

Emitted Notifications

Notification	Description
SEOMENEWNOTIFY	As a result of creating the diagram.
SEPAGENOTIFY	As a result of creating a page.

Create OM Page

Description

The *Create Page* service will create a new page in an existing diagram. The new page will be inserted as the last page in the diagram. The heading and page number symbols exists and are placed automatically relative to the frame symbol. The frame symbol can not be changed to another position.

If a page already exists with the given page name, the service will be ignored and the return status is OK. This can be used for convenience to always send Create Page for all pages including the one already sent in the Create OM Diagram service.

The grid values should be multiples of 5 mm. If the values are zero or less than zero this means that the preferences values are used for the grid values.

A page can be set to be autonumbered. When this is used a page name must be given as well. For consecutive autonumbered pages these names must be “1”, “2”, “3” and so on. It is considered an error if the given page name not is the same as the generated page name when this flag is set.

If the *ShowMeFirst* attribute is set for a new page it will override an already existing ShowMeFirst page.

Tools Supporting the Service

`SET_OME`

Service Request

`SEOMECIFCREATEPAGE`

Parameter	Type	Description
`bufId`	`integer`	Buffer referencing the diagram in which the page will be created.
`pageName`	`string`	Name of the new page in the diagram.
`pageWidth`	`integer`	The width of the page.
`pageHeight`	`integer`	The height of the page.

Parameter	Type	Description
gridWidth	integer	The grid width.
gridHeight	integer	The grid height.
autonumbered	bool	If true the page is an autonumbered page.
showMeFirst	bool	If true this page is set as the Show-MeFirst page.
scale	integer	The scale used when shown, 100 means a 1:1 scale.

Service Reply

SEOMECIFCREATEPAGEREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

```
Invalid diagram buffer id
The pagename is empty
For an autonumbered page the given name must be the
same as the generated pagename: "<gen.pagename>"
```

Emitted Notifications

Notification	Description
SEPAGENOTIFY	As a result of creating a page.

Insert OM Object

Description

The *Insert Object* service will insert an object in a specified position at a specific page. The return value is the object id for the inserted object. Lines connected to the inserted symbol should refer to the objectId returned by this service.

The following parameter checks are performed:

- If a line is inserted the symbols connecting the line must already have been inserted.
- For lines it is an error if there are less than two breakpoints.
- The object type must be a known symbol or line.

The following is a non-exhaustive list of restrictions, applied inside OME when manipulating with symbols, that are **not** checked by this service:

- Positions outside the page area.
- Text syntax for class and instance symbols. However, when an object is inserted with the *updateFlag* set there will be a syntactic check and all eventual syntax errors will be marked in the normal way.
- That lines are connected to symbols with the correct syntax.
- That line endpoints fall on or inside the enclosing rectangle of the symbol it is connected to.
- Overlapping symbols

Tools Supporting the Service

`SET_OME`

Service Request

SEOMECIFINSERTOBJECT

Parameter	Type	Description
bufId	integer	Buffer referencing the diagram in which an object should be shown.
pagename	string	An existing page within the diagram.
objectType	integer	Identify the type of object to insert. Valid values for all symbols are given in sdtsym.h. For lines the values are listed below.
updateFlag	bool	If this flag is true a routine will be called for this page to fix some attributes for the objects. This fixing must be done before the page is saved. If this flag is false this fix will not be done.
object specific parameters	varying	One or more parameters defining the inserted object.

Service Reply

SEOMECIFINSERTOBJECTREPLY

Parameter	Type	Description
status	integer	Service reply status.
objectId	integer	Identifier to the inserted object

Errors

```
Invalid diagram buffer id
Invalid pagename
Invalid fromSymbol id for a line:
Invalid toSymbol id for a line:
Invalid fromSymbol. The fromSymbol is a line:
Invalid toSymbol. The toSymbol is a line:
Number of points in line < 2 :
Invalid objectId for ConnectionPoint line:
Invalid object type:
```

Object Specific Parameters

For each object type the needed parameters are listed below. All coordinates are given in 1/10 mm units in the coordinate system with the upper left corner of the page as origo and the y-coordinate having the positive direction going downwards.

A given coordinate will be adjusted to fall on a grid. For symbols this grid is given by a grid net of 5 x 5 mm. For line breakpoints and text positions the grid net is 2.5 x 2.5 mm.

The *Pagenumber* symbol (objectType number 53) have no extra specific parameters. When inserted there will not be a new symbol created as the pagenumber symbol is always created when the page is created. The text for the pagenumber symbol is generated automatically. The only use of inserting this symbol is to force an update of the page by having the *updateFlag* set to true

- **Class**(57)

Parameter	Type	Description
`left`	`integer`	The x coordinate for the left side.
`top`	`integer`	The y coordinate for the top side.
`nameText`	`string`	The name text part of the symbol.
`attributeText`	`string`	The attribute text part of the symbol.
`operationText`	`string`	The operation text part of the symbol.
`stereotypeText`	`string`	The stereotype text part of the symbol.
`propertiesText`	`string`	The properties text part of the symbol.
`collapsed`	`bool`	If this is true the symbol is collapsed.

- **Instance**(58)

Parameter	Type	Description
left	integer	The x coordinate for the left side.
top	integer	The y coordinate for the top side.
nameText	string	The name text part of the symbol.
attributeText	string	The attribute text part of the symbol.
stereotypeText	string	The stereotype text part of the symbol.
propertiesText	string	The properties text part of the symbol.
collapsed	bool	If this is true the symbol is collapsed.

- **TextSymbol**(28)

Parameter	Type	Description
left	integer	The x coordinate for the left side.
top	integer	The y coordinate for the top side.
text	string	The text in the symbol.
collapsed	bool	If this is true the symbol is collapsed

- **AssociationLine**(113), **AggregationLine**(114)

Parameter	Type	Description
fromID	integer	The objectId for the symbol where the line starts.
toID	integer	The objectId for the symbol where the line ends.
haveName	bool	Flag set if the line has forward name attributes

Parameter	Type	Description
`name`	`string`	Only if *haveName* set: The forward name of the line.
`nameArrow`	`bool`	Only if *haveName* set: Arrow on the forward name.
`XPosition`	`integer`	Only if *haveName* set: The x coordinate for the left top position of the name.
`YPosition`	`integer`	Only if *haveName* set: The y coordinate for the left top position of the name.
`haveRevName`	`bool`	Flag set if the line has reversed name attributes.
`revName`	`string`	Only if *haveRevName* set: The reversed name of the line.
`revNameArrow`	`bool`	Only if *haveRevName* set: Arrow on the reversed name.
`XPosition`	`integer`	Only if *haveRevName* set: The x coordinate for the left top position of the reversed name.
`YPosition`	`integer`	Only if *haveRevName* set: The y coordinate for the left top position of the reversed name.
`haveRoleName`	`bool`	Flag set if the line has role name attributes.
`roleName`	`string`	Only if *haveRoleName* set: The forward role name of the line.
`XPosition`	`integer`	Only if *haveRoleName* set: The x coordinate for the left top position of the forward role name.
`YPosition`	`integer`	Only if *haveRoleName* set: The y coordinate for the left top position of the forward role name.

Parameter	Type	Description
haveRevRoleName	bool	Flag set if the line has reversed role name attributes.
revRoleName	string	Only if *haveRevRoleName* set: The reversed role name of the line.
XPosition	integer	Only if *haveRoleName* set: The x coordinate for the left top position of the reversed role name.
YPosition	integer	Only if *haveRoleName* set: The y coordinate for the left top position of the reversed role name.
haveMultText	bool	Flag set if the line has multiplicity text attributes.
multText	string	Only if *haveMultText* set: The forward multiplicity text of the line.
XPosition	integer	Only if *haveMultText* set: The x coordinate for the left top position of the multiplicity text.
YPosition	integer	Only if *haveMultText* set: The y coordinate for the left top position of the multiplicity text.
haveRevMultText	bool	Flag set if the line has reversed multiplicity text attributes.
revMultText	string	Only if *haveRevMultText* set: The reversed multiplicity text of the line.
XPosition	integer	Only if *haveRevMultText* set: The x coordinate for the left top position of the multiplicity text.
YPosition	integer	Only if *haveRevMultText* set: The y coordinate for the left top position of the multiplicity text.

Parameter	Type	Description
haveQualText	bool	Flag set if the line has qualifier text attributes.
qualText	string	Only if *haveQualText* set: The forward qualifier text of the line.
haveRevQualText	bool	Flag set if the line has reversed qualifier text attributes.
revQualText	string	Only if *haveRevQualText* set: The reversed qualifier text of the line.
haveConstText	bool	Flag set if the line has constraint text attributes.
constraintText	string	Only if *haveConstText* set: The constraint text of the line.
XPosition	integer	Only if *haveConstText* set: The x coordinate for the left top position of the constraint text.
YPosition	integer	Only if *haveConstText* set: The y coordinate for the left top position of the constraint text.
haveOrdered	bool	Flag set if the line has the Ordered attribute.
XPosition	integer	Only if *haveOrdered* set: The x coordinate for the left top position of the Ordered text.
YPosition	integer	Only if *haveOrdered* set: The y coordinate for the left top position of the Ordered text.
haveSorted	bool	Flag set if the line has the Sorted attribute.
XPosition	integer	Only if *haveSorted* set: The x coordinate for the left top position of the Sorted text.

Parameter	Type	Description
YPosition	integer	Only if *haveSorted* set: The y coordinate for the left top position of the Sorted text.
haveRevOrdered	bool	Flag set if the line has the reversed Ordered attribute.
XPosition	integer	Only if *haveRevOrdered* set: The x coordinate for the left top position of the reversed Ordered text.
YPosition	integer	Only if *haveRevOrdered* set: The y coordinate for the left top position of the reversed Ordered text.
haveRevSorted	bool	Flag set if the line has the reversed Sorted attribute.
XPosition	integer	Only if *haveRevSorted* set: The x coordinate for the left top position of the reversed Sorted text.
YPosition	integer	Only if *haveRev* Sorted set: The y coordinate for the left top position of the reversed Sorted text.
derived	bool	Flag set if the line shall be marked as derived.
numPoints	integer	The number of breakpoints for the line. There must be at least 2.
x	integer	x coordinate for a point.
y	integer	y coordinate for a point.

It is checked that symbols exists having the given *fromID* and *toID*. There will be *numPoints x* and *y*-values at the end of the message. Text position and line coordinates are adjusted to the nearest grid point but it will not be checked whether line endpoints are placed on the symbol boundary (the rectangle enclosing the symbol), as they

are expected to be. Extra line segments needed for e.g. a signal route entering a service will be automatically generated.

- **GeneralizationLine**(115)

Parameter	Type	Description
fromID	integer	The objectId for the symbol where the line starts.
toID	integer	The objectId for the symbol where the line ends.
haveName	bool	Flag set if the line has a name attribute.
name	QuotedString	Only if *haveName* set: The name of the line.
XnamePosition	integer	Only if *haveName* set: The x coordinate for the left top position of the name.
YnamePosition	integer	Only if *haveName* set: The y coordinate for the left top position of the name.
numPoints	integer	The number of breakpoints for the line. There must be at least 2.
x	integer	x coordinate for a point.
y	integer	y coordinate for a point.

It is checked that symbols exists having the given *fromID* and *toID*. There will be *numPoints x* and *y*-values at the end of the message. Text position and line coordinates are adjusted to the nearest grid point.

- **Class LinkLine**(123)

Parameter	Type	Description
`fromID`	`integer`	The objectId for the line where the link line starts.
`toID`	`integer`	The objectId for the symbol where the line ends.
`numPoints`	`integer`	The number of breakpoints for the line. Always 2 since no breakpoints are allowed on this line.
`x`	`integer`	x coordinate for a point.
`y`	`integer`	y coordinate for a point.

It is checked that symbols exists having the given *fromID* and *toID*. Observe that the *fromID* must denote an association or aggregation line. Note that *numPoints* always must be 2, and consequently there will be two *x* and *y* coordinates.

Text Editor Services

Show Position

Description

The *Show Position* service makes sure that a certain character specified by line and column is visible in a text editor window:

- If no window is displaying the document identified by bufferId, the buffer is displayed in a window according to the user's window re-use policy.
- A window is brought to the front.
- The text in the window is re-centered so that the indicated character is displayed.

Note that the actual display position after re-centering is undefined if the line and/or column values do not make sense.

Tools Supporting the Service

SET_TE

Service Request

SETESHOWPOSITION

Parameter	Type	Description
bufid	integer	Identifies the buffer
line	integer	Indicates the line that should be shown
column	integer	Indicates which character on the line which should be shown

Service Reply

SETESHOWPOSITIONREPLY

Parameter	Type	Description
status	integer	Service reply status.

Errors

Buffer id does not exist

Select Text

Description

The *Select Text* service selects a range of text in a window displaying a given text buffer.

- If no window is displaying the document identified by bufferId, the buffer is displayed in a window according to the user's window re-use policy.
- A window is brought to the front.
- The text in the window is re-centered so that the indicated character is displayed.

Note that the actual selection is undefined if one or more of the line and/or column values do not make sense.

Tools Supporting the Service

`SET_TE`

Service Request

`SETESELECTTEXT`

Parameter	Type	Description
`bufid`	`integer`	Identifies the buffer
`line1`	`integer`	Indicates where the selection should start
`column1`	`integer`	Indicates the character where the selection should start.
`line2`	`integer`	Indicates the line where the selection should end.
`column2`	`integer`	Indicates the character where the selection should end.

Service Reply

`SETESELECTTEXTREPLY`

Parameter	Type	Description
`status`	`integer`	Service reply status.

Errors

`Buffer id does not exist`

Notifications

The *notifications* inform the environment when something significantly happens. Notifications are only available as PostMaster messages.

To receive a notification one must subscribe on that particular notification (or on all notifications). This is accomplished either statically, by an entry in an additional configuration file, or dynamically by the services *Add Tool* and *Add Tool Subscription*.

Overview of Available Notifications

Tool notifications

Message	Description
SESTARTNOTIFY	When a tool is started.
SESTOPNOTIFY	When a tool is stopped.

Diagram notifications

Message	Description
SELOADNOTIFY	When a diagram is loaded.
SEUNLOADNOTIFY	When a diagram is unloaded.
SEDIRTYNOTIFY	When a diagram becomes modified.
SESAVENOTIFY	When a diagram is saved.
SESDLENEWNOTIFY	When an SDL diagram is created.
SEMSCENEWNOTIFY	When an MSC diagram is created.
SEOMENEWNOTIFY	When an OM diagram is created.
SETENEWNOTIFY	When a text diagram is created.

Start Notify

Description

This message is broadcast when a new tool is started. That is, when it connects to the PostMaster. It is sent automatically by the SPInit function.

Tools Issuing the Notification

`The started tool`

Notification

`SESTARTNOTIFY`

Parameter	Type	Description
`toolType`	`integer`	The kind of tool that was started.
`argv0`	`string`	The filename (with a complete path) of the started tool as obtained by reading `argv[0]`.

Stop Notify

Description

This message is broadcast when the tool disconnects from the PostMaster and terminates.

Tools Issuing the Notification

`The tool which stops`

Notification

`SESTOPNOTIFY`

Parameter	Type	Description
`toolType`	`integer`	The kind of tool that stopped.

Load Notify

Description

Broadcast when the diagram is loaded in an editor.

Tools Issuing the Notification

```
SET_SDLE
SET_MSCE
SET_OME
SET_TE
```

Notification

```
SELOADNOTIFY
```

Parameter	Type	Description
bufId	integer	Buffer id of the loaded diagram.
fileName	string	Name of the loaded file.
diagramType	integer	Diagram type of the loaded diagram.
diagramName	string	Diagram name of the loaded diagram.

Unload Notify

Description

Broadcast when the diagram is unloaded in the editor. No assumptions could be made whether or not the diagram was saved. The buffer id of the unloaded diagram is then not longer valid and cannot be used anymore.

Tools Issuing the Notification

```
SET_SDLE
SET_MSCE
SET_OME
SET_TE
```

Notification

```
SEUNLOADNOTIFY
```

Parameter	Type	Description
bufId	integer	Buffer id of the unloaded diagram.

Dirty Notify

Description

Broadcast when a diagram becomes dirty, i.e. when the user has modified the diagram.

Tools Issuing the Notification

```
SET_SDLE
SET_MSCE
SET_OME
SET_TE
```

Notification

```
SEDIRTYNOTIFY
```

Parameter	Type	Description
bufId	integer	Buffer id of the diagram that became dirty.

Save Notify

Description

The message is broadcast when a diagram is saved.

Tools Issuing the Notification

```
SET_SDLE
SET_MSCE
SET_OME
SET_TE
```

Notification

```
SESAVENOTIFY
```

Parameter	Type	Description
bufId	integer	Buffer id of the saved diagram.
fileName	string	The filename in which the diagram was saved.

SDL New Notify

Description

Broadcast when a diagram is created in the SDL Editor. In this case a *Load Notify* is not broadcast.

Tools Issuing the Notification

SET_SDLE

Notification

SESDLENEWNOTIFY

Parameter	Type	Description
bufId	integer	Buffer id of the new diagram.
virtuality	string	Kind of virtuality
diagramType	integer	Type of diagram.
diagramName	string	Name of the diagram.
qualifier	string	A qualifier for the new diagram. The qualifier is not a full qualifier, since it does not include the diagram path to the root diagram.

MSC New Notify

Description

Broadcast when a diagram is created in the MSC Editor. In this case a *Load Notify* is not broadcast.

Tools Issuing the Notification

SET_MSCE

Notification

SEMSCENEWNOTIFY

Parameter	Type	Description
bufId	integer	Buffer id of the new diagram.
diagramName	string	Name of the new diagram.

OM New Notify

Description

Broadcast when a diagram is created in the OM Editor. In this case a *Load Notify* is not broadcast.

Tools Issuing the Notification

SET_OME

Notification

SEOMENEWNOTIFY

Parameter	Type	Description
bufId	integer	Buffer id of the new diagram.
diagramName	string	Name of the new diagram.

TE New Notify

Description

Broadcast when a diagram is created in the Text Editor. In this case a *Load Notify* is not broadcast.

Tools Issuing the Notification

SET_TE

Notification

SETENEWNOTIFY

Parameter	Type	Description
bufId	integer	Buffer id of the new diagram.
diagramName	string	Name of the new diagram.

Auxiliary Messages

Communicating with Simulators

Description

SESDLSIGNAL is sent by an SDL simulator when it sends an SDL signal to its SDL environment. Other simulators (or any applications connected to the PostMaster) can then receive the message and send back messages of the same type.

The PIds consist of a UNIX PId **(on UNIX)**, or an identifier ”pid” known by the PostMaster **(in Windows)**, in both cases an integer, and an SDL process instance number (a hex number), which normally is used only in comparisons. The PIds may be used to uniquely identify a single receiver if several receivers exist, but can be ignored if only two tools are communicating

Tools Issuing and Recognizing the Message

SET_SDLENV

Message

SESDLSIGNAL

Parameter	Type	Description
signalName	integer	SDL name of the sent signal
globalReceiverPid	integer	See above.
localReceiverPid	string	See above.
globalSenderPid	integer	See above.
localsenderPid	string	See above.
signalParam	string	Sea above.

PostMaster Operation Failed

Description

SEOPFAILED is sent by the PostMaster when an internal problem inhibited a message to be handled properly. Three major categories of errors exists:

- Communication error, i.e. the underlying network (e.g. UNIX socket) reported an error
- An explicit address did not exist
- An operation requested from the PostMaster failed.

Tools Issuing and Recognizing the Message

SET_POST

Message

SEOPFAILED

Parameter	Type	Description
message	integer	The message causing the error.
severity	integer	Kind of error.
pid	integer	The PId of the subscriber (if a specific subscriber was involved).
errorCode	integer	The error itself.
messageParam	string	The remaining information is the data (or the first part of the data) of the message causing the error.

Chapter 11 The Public Interface

Chapter

12

Using the Public Interface

This chapter contains a number of examples of how the use the features provided by the Public Interface.

For a reference to the topics that are exemplified in this chapter, see *chapter 11, The Public Interface*.

Introduction

The first example shows a simple application, the *Service Encapsulator*, by which it is possible to request SDL Suite or TTCN Suite tools services from the Operating Systems command line. The example shows how to use the PostMaster interface.

The second example uses the Service Encapsulator and shows the usage of some SDL Suite Services. Since the “DOS” command prompt is rather weak in its scripting capabilities, no example script file is included **for Windows** in this example.

The third example uses the Service Encapsulator and shows the usage of some TTCN Suite Services.

The final example exemplifies how to integrate an SDL simulator and a separate user interface **(UNIX only)**. How to design such a user interface is described in a detailed example. It is assumed that the reader has experience of creating and running SDL simulators.

The Service Encapsulator

Introduction to the Service Encapsulator

This section shows an application, the Service Encapsulator, implementing a command line service request encapsulation. That is, services as made available by the Public Interface, could be obtained from the command line.

The purpose of this example is to show:

- How the PostMaster interface is used when working with services in SDL Suite and TTCN Suite.
- By using the Service Encapsulator, show the usage of a number of services.

The application is general in the sense that it does not require any SDL Suite or TTCN Suite tool to be running (only the PostMaster is required to be running), but in this context it exemplifies a number of services.

From the outside world the application behaves like a tool providing *remote procedure call* functionality. That is, when the service is request-

ed, the application synchronously waits for the service reply, before returning (the application exits).

The example is found in the SDL Suite and TTCN Suite distribution in the subdirectory `examples` and then `public_interface`.

The Service Encapsulator is also available as a tool in the SDL Suite and TTCN Suite environment, see *"The Service Encapsulator" on page 537 in chapter 11, The Public Interface*.

Design

In this section is a few important aspects in the design and implementation of the *Service Encapsulator* given. The source file is named `serverpc.c` and the executable is named `serverpc`.

The tool connects to the PostMaster by issuing:

```
SPInit(SET_EXTERN, argv[0], spMList) (in SDL Suite)
SPInit(IET_EXTERN, argv[0], ipMList) (in TTCN Suite)
```

The first parameter gives the type of application. In this case `SET_EXTERN` or `IET_EXTERN` is used. This application type is a predefined tool type for external applications connecting to SDL Suite and TTCN Suite. The variable `spMList` or `ipMList` gives the possibility to convert between textual (symbolic) values and integers.

If `SPInit` succeeds, `SPInit` broadcasts a `SESTARTNOTIFY`, indicating to all other tools in the environment that an additional tool is connected.

```
if ((tool=atoi(argv[1])) == 0 )
  tool = SPConvert(argv[1]);
```

If the parameter `<tool>` was given textually, the function `SPConvert` converts it to the corresponding integer value. If a mapping cannot be found, -1 is returned. This is an error and will be detected in the `SPSendToTool` function, see below.

```
if ((sendEvent=atoi(argv[2])) == 0 )
   sendEvent = SPConvert(argv[2]);
```

Allocating memory for the message parameter is done as follows:

```
for (i=3;i<argc;i++) {
  p = realloc(p, strlen(p) + strlen(argv[i]) + 2);
  if (strlen(p)>0)
    p = strcat(p, " ");
  p = strcat(p, argv[i]);
}
```

The `handleescape` function allows carriage return "`\n`", "`\r`" and "`\0`" to be used when sending messages:

```
p = handleescape(p);
```

A normal service request is issued by:

```
status=SPSendToTool(tool, sendEvent, p, strlen(p)+1;
```

The last parameter tells the length of the parameter provided by `argv[3]` and must be extended by 1 to include the terminating `\0` character.

Now the service request message is sent to the PostMaster which forwards it to the server application. If not busy, the Service application starts to process the service request and when finished it replies with a service reply message to the issues. However, during the processing other messages can be sent or broadcast to the *Service Encapsulator* (the service issuer). Therefor we must enter a loop where we stay until the reply message is received. Other kind of messages are dropped.

```
do {
   if (( status=SPRead(SPWAITFOREVER, &pid,
         &replyEvent,(void*)&replyMessage, &len))!=0
)
      OnError("Error reading from postmaster: ");
} while ( ! SP_MATCHREPLY(sendEvent, replyEvent) );
```

The function `SPRead` reads a message form the PostMaster. The first parameter, the timeout, could be set to wait indefinitely until a message arrives, using the value `SPWAITFOREVER`.

This simple application has a drawback: if something unexpectedly happens to the server, it might be that no service reply will be sent which means that we block in `SPRead` forever. The behavior could be enhanced by setting a timeout, and when the timeout expires, check the state of the server (Use the PostMaster service Get Tool Pid).

If the service `readattribute` was issued, special care must be taken to handle the answer, since it contains binary data. The length of the data part ends the `replyMessage`. The end is indicated by an ASCII 0. The

data part immediately follows the ASCII 0 character and might contain non-ASCII characters. Our printout assumes ASCII data.

```
if ( sendEvent == SEREADATTRIBUTE ) {
   printf("%s", replyMessage+1);
  printf("%s\n", replyMessage+ strlen(replyMessage)
          +1);
}
```

The macro `SP_MATCHREPLY` compares the issued service request message and the reply message and returns true if the correct received message was the correct service reply message.

The service reply is then printed on standard output. The following prints the service reply data:

```
int i;
for( i=2;i<len;i++ )
  putchar(replyMessage[i]);
putchar('\n');
```

Counting starts at position 2, omitting the service reply status.

```
free (replyMessage);
```

`replyMessage` was dynamically allocated using malloc in `SPRead`, so it must be freed.

Before terminating the application, notify all other applications in the environment that it is going to break its connection to the PostMaster

```
SPBroadcast(SESTOPNOTIFY, NULL, 0);
```

Finally we terminate the PostMaster connection by calling:

```
SPExit ();
```

Generating the Application for the SDL Suite

To generate the application do the following:

1. Change the current directory to the sub directory containing the example:

   On UNIX: `cd <Installation Directory>/examples/public_interface`

   In Windows: `cd <Installation Directory>\examples\public_interface`

2. Set the environment variable `telelogic` to point to the right directory. This is platform dependent.

   On UNIX: `setenv telelogic <Installation Directory>`

   In Windows: `set telelogic=<Installation Directory>`

3. Generate the application:

   On Solaris: `make -f Makefile.solaris`

   In Windows using Microsoft: `nmake /f Makefile.msvc`

Using the SDL Suite Services

Introduction

This section exemplifies the usage of some SDL Suite services. The examples take advantage of the *Service Encapsulator* described in the previous section.

In these examples, we only use textual values of the parameters `<tool>` and `<service>`. Available textual values is found in the `spMList` variable in `sdt.h`.

In the examples below, the *Service Encapsulator* is executed from a `csh` shell **(on UNIX)**, or from the "DOS" prompt **(in Windows)**. If another shell is used, the examples below might need to be modified in order to supply the parameters correctly. Take care how to supply the quoted strings.

> **Note:**
>
> Due to limitations in DOS, this Service Encapsulator example is not as extensive in Windows as the UNIX version.

Load External Signal Definitions into the Information Server

For this service to be available, the Information Server must be started. This could be done by either starting the Type Viewer from the Organizer, by requesting *Signal Dictionary* support in the SDL Editor or by using the *Start service*. For a complete service description, See *"Start Tool" on page 539 in chapter 11, The Public Interface*.

Requesting the Service

It is assumed that a file `a.pr` is to be loaded. Note that the file to be loaded must be specified with a full directory path.

On UNIX: `serverpc info loaddefinition /usr/ab/a.pr`

In Windows: `serverpc info loaddefinition c:\sdt\ex\a.pr`

If the Information Server is started, the following reply message is returned:

```
Reply status is OK
```

If the Information Server is not started, we get the following reply message.

```
Error sending to postmaster:  server not connected
to postmaster
```

The external files loaded into the SDL Infoserver should have the following appearance e.g.:

```
signal sig1
signal sig2
signal sig3
```

Obtain Source (SDL/GR Reference)

Returns a complete SDT reference of each of the selected symbols in the specified editor.

```
serverpc sdle obtainsource
```

If the editor was already started and there was a selection the reply message might look like:

```
Reply status is OK
 1
"#SDTREF(SDL,c:\IBM\Rational\SDL_TTCN_Suite6.3\sdt\e
xamples\demongam\demon.spr(1),125(30, 70))"
```

Note that the reference returns a complete file path of the diagram file in which the selection was made.

If the editor did not contain any selection, the reply becomes:

```
Reply status is OK
```

```
0
```

For a complete description of the service, see *"Obtain GR Reference" on page 599 in chapter 11, The Public Interface*.

Show Source

Selects the object given by the parameters in an editor. The editor is started if necessary as a side effect. A system must however be opened in the Organizer containing the specified reference.

To test this service we could now deselect all selections in the SDL Editor and use the reference extracted by `obtainsource` to select this symbol again. Note how the SDT reference is quoted to pass it from the shell into the tool.

On UNIX:
```
serverpc organizer showsource \
  """#SDTREF(SDL,/usr/sdt-inst/examples/simulator- \
  integration/sunos4/demon.spr(1),125(30,70))"""
```

In Windows:
```
serverpc organizer showsource
\"#SDTREF(SDL,c:\IBM\Rational\SDL_TTCN_Suite6.3\sdt\
examples\demongam\demon.spr(1),125(30, 70))\"
```

If the reference is found the following message is replied:

```
Reply status is OK
```

For a complete description of the service, see *"Show Source" on page 598 in chapter 11, The Public Interface*.

For a complete description of how to specify a SDT reference, see "Syntax" on page 917 in chapter 18, *SDT References*.

Dynamic Menus

The following example shows how a dynamic menu is created and how a number of menu items are inserted in the dynamic menu. The script `dyn-menu` in the example directory performs the example below.

For a complete description of the services, see "Add Menu" on page 581, "Delete Menu" on page 582, "Clear Menu" on page 583 and *"Add Item to Menu – Graphical Editors" on page 593 in chapter 11, The Public Interface*.

Add a new menu to the SDL Editor:

```
serverpc sdle menuadd \"dyn-menu\"
```

Reply:

```
Reply status is OK
```

Add a menu item which executes the OS command `ls` **(on UNIX)** or `DIR` **(in Windows)** after a confirmation from the user:

On UNIX:
```
serverpc sdle menuadditem  \
\"dyn-menu\" \"ls\" 0 \
\"Perform OS command ls\"  0 0 0 \
\"OK to perform ls!\" 1 1 \"ls\"
```

In Windows:
```
serverpc sdle menuadditem \"dyn-menu\"
\"DIR\" 0 \"Perform OS command DIR\" 0 0 0 \"OK to
perform OS command DIR!\" 1 1 \"CMD /C DIR /W\"
```

Reply:

```
Reply status is OK
```

Then a menu item is added displaying the SDT reference of the selected symbol on standard output **(on UNIX)**, or in the Organizer Log window **(in Windows)**:

On UNIX:
```
serverpc sdle menuadditem  \
\"dyn-menu\" \"SDT-ref\" 1 \
\"export SDT ref\"  0 0 1 \
\"\" 1 1 \"echo \'%g\'\"
```

In Windows:
```
serverpc sdle menuadditen \"dyn-menu\"
\"SDT-ref\" 1 \"export SDT-ref\" 0 0 1 \"\" 1 1
\"CMD /C echo %%g\"
```

Reply:

```
Reply status is OK
```

Note that only dimmed if more than one item is selected. Normally a selected symbol includes a selection of in-going and out-going flowlines. These flowlines must be de-selected in order to get the SDT reference of the symbol.

Finally display the file containing the selected symbol on standard output **(on UNIX)**, or in the Organizer Log window **(in Windows)**:

On UNIX:
```
serverpc sdle menuadditem  \
\"dyn-menu\" \"filename\" 0 \
\"export filename\"  0 0 1 \
\"\" 1 1 \"ls -ls %a\"
```

In Windows:
```
serverpc sdle menuadditem \"dyn-menu\"
\"filename\" 0 \"export filename\" 0 0 1 \"\" 1 1
\"CMD /C DIR %%a\"
```

Reply:

```
Reply status is OK
```

Extended Object Data Attributes (UNIX only)

The SDL Suite allows external attributes to be added to SDL Suite objects. These extensions are persistent. That is, they are stored in the normal SDL diagram files and could be used at subsequent sessions. This **UNIX only** example gives a brief introduction to such extended attributes.

An description of extended attributes are found in "Extended Data Attribute" on page 617 in chapter 11, *The Public Interface*. Extended attributes are handled as binary data by the SDL Suite. As such, their services are preferably accessed via C program interface, in which binary data is easily handled.

However, in this example extended attribute services are accessed via the *Service Encapsulator*, which works for simple examples.

The example assumes one selection in an SDL Editor. This reference is saved in a shell variable

```
set ref=`serverpc sdle obtainsource`
```

If we have exactly one selection, we set the extended attribute. Parameter 9 will give us the SDT reference.

```
serverpc sdle createattribute 0 0 $ref[9] 0 \
  \"myComment\" \"\"  6"\0"MyData
```

Note that the length of data (`MyData)` must be manually calculated and that the data part must be preceded by a "\0".

We get the reply:

```
Reply status is OK
```

Now we extract the extended attribute on the selection.

```
serverpc sdle readattribute 0 0 $ref[9]
```

Which replies:

```
Reply status is OK
0 "myComment" "" 6MyData
```

Note that data immediately follows its' length without any spaces. In the reply message, there is a ASCII 0 character after the length, preceding the data part.

Then we update the attribute by;

```
serverpc sdle updateattribute 0 0 $ref[9] 0
\"newComment\" \"\" 7"\0"NewData
```

Reply:

```
Reply status is OK
```

Finally we read the extended attribute and should receive the updated value.

```
serverpc sdle readattribute 0 0 $ref[9]
```

Which replies:

```
Reply status is OK
0 "NewComment" "" 7NewData
```

The script `extended-attributes` performs the above described actions.

Using TTCN Suite Services

This section is meant to exemplify the usage of a few TTCN Suite services. The examples are made using the Service Encapsulator described in a previous section. For the complete list of supported services, see "TTCN Suite Services" on page 554 in chapter 11, *The Public Interface*.

Opened Documents

This command retrieves a list of information about the opened TTCN documents.

```
serverpc itex openeddocuments
```

This will yield a list of open TTCN documents (not necessarily shown) that looks like this (see complete list for description of the list format):

```
Reply status is OK
Doc1:1:/path/doc1.itex:/path/#doc1.itex
Doc2:2:/path/doc2.itex:/path/#doc2.itex
```

If the TTCN environment is not yet started the message will be:

```
Error sending to postmaster: server not connected to
postmaster
```

Find Table

To use this command the TTCN document in which the table is located must be open.

To invoke the service the following Service Encapsulator command is made:

```
serverpc itex findtable 1 \"Foo\"
```

which will search for the table named `Foo` in the open TTCN document with buffer identifier 1 and open it the Table Editor.

If the table is found the table will be shown and the following message will be returned:

```
Reply status is OK
```

If the table can not be found, an error message is issued:

```
Unable to find object 1 Foo
```

Integrating Applications with SDL Simulators

An example of utilizing the functionality of the PostMaster is to connect a simulator generated by the SDL Suite to another application, typically a user interface (UI). The section "Example of Use (UNIX only)" on page 692 presents a detailed description of such an example **(UNIX only)**. The sections below concentrate on overall design issues and serves as an introduction to that example.

The communication between an SDL simulator and another application is handled by the PostMaster, and can be seen as occurring on two levels:

- Sending and receiving SDL signals
- Sending and receiving PostMaster messages, containing the SDL signals

These two levels of communication are described further below.

Transferring SDL Signals

A simulator communicates with the world outside by sending and receiving SDL signals to/from its environment. For another application to communicate with the simulator, it must also be able to interpret those SDL signals, and to send and receive them. This includes mapping the information contained in the SDL signal (name and parameters) to components and actions in the UI, e.g. invoking a command or changing the contents of an output field.

To ensure successful communication with an SDL simulator, the SDL signal interface to the environment should be designed with regard to the connected applications. Decisions made when designing a UI can influence the design of the simulator interface, and vice versa. It is important to have a clearly defined signal interface to the applications that will communicate with the simulator.

Transferring PostMaster Messages

The PostMaster communicates with different tools by sending messages of a defined format. The SDL signals must therefore be transformed to PostMaster messages before they can be transferred between the tools. A few predefined PostMaster messages are available for the purpose of handling SDL signals.

Each tool designed to communicate by using SDL signals must have an interface to the PostMaster that handles the transformation to and from an appropriate message.

In the case of an SDL simulator, this interface is generated automatically. To invoke the transformation, so that SDL signals to and from the environment are transferred using the PostMaster, the monitor command Start-SDL-Env is to be used.

For other tools, the interface to the PostMaster must be implemented separately. The interface must use functions described later in this section to connect to the PostMaster, and to send and receive PostMaster messages containing SDL signals. This includes packing the information contained in the SDL signal (name and parameters) so that it can be sent using a PostMaster message, and unpacking the same information of a received message. See *"Input and Output of Data Types" on page 2137 in chapter 49, The SDL Simulator* for more information on suitable data formats.

Example of Use (UNIX only)

This section describes an example of how to connect a user interface (UI) to an existing SDL simulator. The simulated system used in the example is the well-known Demon game, described in "The Demon Game" on page 41 in chapter 3, *Tutorial: The Editors and the Analyzer, in the SDL Suite 6.2 Getting Started*. All necessary code for the example is provided with the release.

The DemonGame simulator is connected through the PostMaster to a control program, consisting of two parts; an interface to the PostMaster and a user interface (see Figure 162).

The PostMaster interface establishes a connection to the PostMaster and communicates with the DemonGame simulator. Messages are sent via the PostMaster to the simulator where they are seen as SDL signals coming from the environment. Correspondingly, SDL signals to the en-

vironment are sent via the PostMaster as messages back to the PostMaster interface.

The UI facilitates sending and receiving of the SDL signals sent to and from the environment in the Demon game. A command-based UI (written in C) is implemented. However, the ideas are general and could be used if a graphical user interface is to be used. The UI must be able to forward the SDL signals Newgame, Endgame, Probe and Result, and to present the SDL signals Win, Lose and Score.

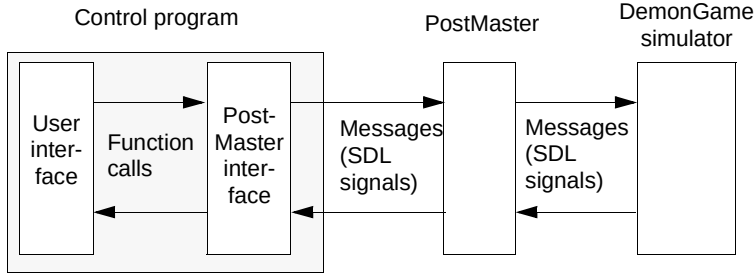


Figure 162: System parts and their communication

Note:

The simulator and the control program have the **same communication interface** to the PostMaster. The simulator's interface to the PostMaster is **generated** automatically by the SDL Suite.

All C functions, C preprocessor symbols and PostMaster messages used in this example are described in "PostMaster Reference" on page 490 in chapter 10, *The PostMaster*.

The PostMaster Interface

All communication with the DemonGame simulator is handled by the interface to the PostMaster, a C program called `env.c`. This program contains the functions `Init`, `Send_to_PM`, `Receive` and `Exit_PM`, all of which must be called by the UI. Please note that the error handling in this program is very simple and does not comply with the design recommendations mentioned above.

Functions in env.c

- `Init` establishes a connection to the PostMaster by calling `SPInit`, identifying this tool as an SDL environment. The function broadcasts the message `SESTARTNOTIFY` to inform other tools that this tool has started. Finally it calls `Init_UI`, a function in the UI.
- `Send_To_PM` takes the name of an SDL signal as parameter and broadcasts this signal as an `SESDLSIGNAL` message to the simulator.
- `Receive` calls `SPRead` to wait for messages from the PostMaster, and acts suitably:
  - If the message contains an SDL signal from the simulator (Score, Lose or Win), it is passed on to the UI by calling `Send_To_UI`, a function in the UI to present the receipt of the signal. It also returns true in these cases.
  - If the message is a request to stop executing (`SESTOP`), it is accepted by calling `Exit_UI`, a function in the UI that in turn calls `Exit_PM`.
  - If the message is `SEOPFAILED`, the PostMaster has failed to handle a message properly. You should then decode the message's parameters by calling
- `Exit_PM` broadcasts the message `SESTOPNOTIFY` and breaks the connection to the PostMaster by calling `SPExit`

Command-Based User Interface

The command-based UI prompts for and recognizes the commands "Newgame", "Endgame", "Probe", and "Result", corresponding to the SDL signals. When any of the SDL signals Score, Win or Lose are received, this information is printed. The command "Exit" exits the program.

The C program for the command-based UI is called `command.c`. It contains the functions:

- `main`
- `GetInput`
- `Appl_Receive`
- `Init_UI`
- `Exit_UI`
- `Send_To_UI`

Functions in command.c

- `main` calls `Init` in the PostMaster interface, prompts for input and examines the reading state of the file descriptors for the keyboard and the PostMaster message port. If input comes from the keyboard, `GetInput` is called and any valid SDL signal is sent by calling `Send_To_PM` in the PostMaster interface. If input comes from the message port, the message is read by calling `Appl_Receive`.
- `GetInput` reads the keyboard and compares the entered string with the available commands. If it is an SDL signal, the function returns true. If "exit" is entered, `Exit_UI` is called.
- `Appl_Receive` simply calls `Receive` in the PostMaster interface. It is included to make the functional interfaces equivalent for both user interfaces.
- `Init_UI` sets a file descriptor for the PostMaster message port and initializes unbuffered terminal output.
- `Exit_UI` calls `Exit_PM` in the PostMaster interface and exits the program.
- `Send_To_UI` prints the name of the received signal and any signal parameters on the terminal.

The figures below show how the different functions and other parts in the DemonGame UI communicate with each other.

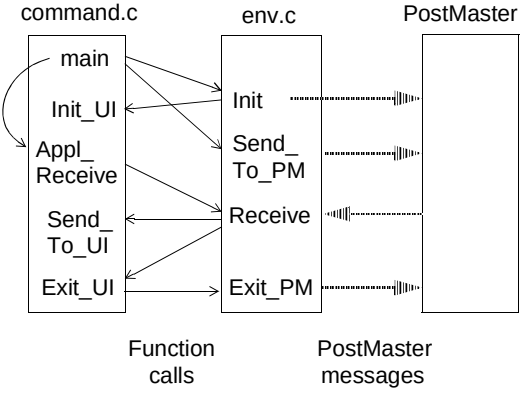


Figure 163: Communication diagram

Running the Example

To be able to build and execute the example program, the software provided on the distribution media must first be properly installed.

Building the Tools

This section describes how to build the tools that will communicate via the PostMaster, i.e. the DemonGame simulator and the UI to the simulator.

The tools are provided in source code form and in binary form for the architecture on which you have purchased the SDL Suite.

The DemonGame Simulator

In the original implementation of DemonGame provided on the installation CD (in the `examples/demongame` directory), a design error has deliberately been introduced into the game definition. See *"Dynamic Errors" on page 153 in chapter 4, Tutorial: The SDL Simulator, in the SDL Suite 6.2 Getting Started* for more information.

For the example program to execute correctly, this error must be corrected. New diagrams for the block GameBlock (`gameblock.sbk`) and the process Main (`main.spr`) are therefore provided, and are to be used to generate a new simulator. You can find the necessary files in the directory named `sdt/examples/simulatorintegration`.

1. Either replace the above diagrams in the original DemonGame implementation, or copy the DemonGame diagrams to a new directory.
2. Re-generate the simulator from the Organizer in the usual way.
3. Change to the directory containing the files `env.c`, `command.c`, and `makefile`.
4. Build the application with the UNIX `make` command. An executable named `command` is now created.

Executing a Session

This section describes how to start the communicating tools and initiate a session.

1. Make sure you have the SDL Suite running, so that an instance of the PostMaster is started.
2. Start the newly generated DemonGame simulator from the Organizer.
3. Give the following commands to the simulator, preferably by including the provided command file `init.sim`.

   `Start-SDL-Env`

   Start handling of the SDL environment. See *"Start-SDL-Env" on page 2178 in chapter 49, The SDL Simulator*.

   `Set-Trace 1`

   Restrict trace output to show only signals sent to and from the SDL environment.

   `Go`

   Start executing the simulation.
4. Start the command-based UI (`command`) from the UNIX prompt.
5. Send SDL signals from the UI by entering the signal name or pushing the corresponding button. (See the description earlier in this section.) Note the trace output in the simulator and the output in the UI.
6. Exit by entering "exit" or using the *File* menu.

Chapter 12 Using the Public Interface

Chapter

13 *The ASN.1 Utilities*

The ASN.1 Utilities perform three main functions:

- **They can translate an ASN.1 module to an SDL package. This makes it possible to use ASN.1 types and values in SDL.**
- **They make it possible for the TTCN Suite to retrieve external ASN.1 types and values that are used in TTCN.**
- **They produce type information for BER coders in SDL.**

> **Note: ASN.1 support in the TTCN Suite**
>
> **The TTCN to C compiler supports only a limited subset of ASN.1. See "TTCN ASN.1 BER Encoding/Decoding" on page 40 in chapter 1, *Compatibility Notes, in the Release Guide* for further details on the restrictions that apply.**

This chapter is the reference manual for the ASN.1 Utilities.

Introduction

This chapter describes the ASN.1 Utilities. It is assumed that the reader is familiar with ASN.1.

Application Areas for the ASN.1 Utilities

The main foreseen applications of the ASN.1 Utilities are the following:

- A lot of telecommunication protocols and services are defined using ASN.1. The ASN.1 Utilities make it easier to specify and implement these with SDL.
- The ASN.1 Utilities enable the SDL Suite and the TTCN Suite to share common data types by specifying these in a separate ASN.1 module.
- The ASN.1 Utilities generate type information for BER encoding/decoding for the SDL Suite.

Overview of the ASN.1 Utilities

The ASN.1 Utilities support the following main functions:

1. Perform syntactic and semantic analysis of ASN.1 modules.
2. Generate SDL code from ASN.1 modules.
3. Extract the ASN.1 types and values which are referred in the TTCN Suite.
4. Generate type information for BER encoding and decoding for the SDL Suite.

For further information about BER encoding and decoding, see chapter 58, *ASN.1 Encoding and De-coding in the SDL Suite, in the User´s Manual*.

In normal cases, the ASN.1 Utilities are completely hidden for the user by the SDL Analyzer and the TTCN Analyzer.

From the user's point of view, an ASN.1 module is very similar to an SDL package: ASN.1 data types can be defined in a module, and then be used within SDL, using operators that are defined in ITU Recommendation Z.105. When an SDL system containing ASN.1 modules is

analyzed, the Analyzer will order the ASN.1 Utilities to translate these modules into corresponding SDL packages.

In the TTCN Suite, indirect use of the ASN.1 Utilities is made by the ASN.1-by-reference table. When such a table is analyzed, the TTCN Suite orders the ASN.1 Utilities to extract the ASN.1 types and values in a specified ASN.1 module. For more information about this functionality, see "ASN.1 External Type/Value References" on page 1197 in chapter 26, *Analyzing TTCN Documents (on UNIX)*.

Using the ASN.1 Utilities

The ASN.1 Utilities are implemented in the executable `asn1util`. `asn1util` can be used in two ways:

1. Stand-alone from the organizer (command-line interface).
2. Via the PostMaster

Command-Line Interface

Usage: `asn1util [options] { <file> [options] }*`

Option	Meaning
`-h`	display a help message
`-v`	display version
`-q`	be quiet, suppress some output messages
`-c`	generate encode/decode type information for the SDL Suite and asn1_cfg.h configuration file
`-g`	generate coder information for TTCN Suite
`-B`	set BER as default encoding
`-P`	set PER as default encoding
`-N <name>`	set <name> as default encoding
`-m`	include module name in encode/decode type nodes and macros

Option	Meaning
`-n <name>`	use <name> for the name of the interface (*.ifc) files
`-S <config>`	use <config> for type names configuration in (*.ifc) files
`-s <file>`	generate SDL output in `<file>`
`-a`	append the output to an existing file instead of creating a new file
`-b`	generate SDL body only, i.e. do not generate package headings (makes it possible to import generated SDL with #INCLUDE)
`-r`	generate references (#SDTREF) to source file
`-e`	generate all operators for the SDL enumerated type as listed in Z.105. Default is to emit some of the operators in Z.105
`-O`	generate values for SDL Make operator with optional and default support
`-u <package>`	add “use <package>;“ to all generated SDL packages
`-J <name> <files>`	Join all ASN.1 modules from <files> into one SDL package <name> (see Example 32 on page 703)
`-K <file>`	Perform substitution for keywords listed in <file> (see “Keywords substitution” on page 705)
`-i <file>`	generate TTCN output to <file>
`-l <file>`	take command line from <file>
`-post`	wait for commands via the PostMaster (see “PostMaster Interface” on page 704)
`-T<dir>`	put generated code in directory `<dir>`

Example 30 ----------

```
asn1util -r -s myfile.pr -c myfile.asn
(myfile.asn contains ASN.1 module MyModule)
```

The command in the example translates the module MyModule in file myfile.asn to an SDL package MyModule in file 'myfile.pr'. The generated package will contain backward references to the source file 'myfile.asn'. Encode/decode type nodes are generated in C-source file 'MyModule_asn1coder.c' and C-header file 'MyModule_asn1coder.h'. A configuration file “asn1_cfg.”' with compile switches for coder related files is generated.

Example 31 ----------

```
asn1util AsnModule1.asn AsnModule2.asn
```

If no options are specified, then asn1util only performs syntactic and global semantic analysis for AsnModule1.asn and AsnModule2.asn, no output is generated.

If no input file is specified, then asn1util does nothing except showing help or version number if correspondent options are specified.

Example 32 Joining modules ----------

```
asn1util -J Join-Module -s my.pr my1.asn my2.asn
```

The ASN.1 modules from the files my1.asn and my2.asn will be joined together in the SDL package Join_Module. Name clashes may occur if the same name is available in different ASN.1 modules the filed are joined. These problems are resolved according to a set of name clash resolving rules, see “Joining modules” on page 708.

Configuration file generation

For the `-c` option encode and decode type information is generated to C-files. Also asn1util performs the analysis of ASN.1 types and some features used in the specification and generates file `asn1_cfg.h`.

This file contains compile switches that are referenced from inside the coders code. When `asn1_cfg.h` is used by the build process the preprocessor automatically throws away useless parts of the code from encod-

ing and decoding related files. This helps to reduce the code size and improve the performance of encoding and decoding procedures.

For example, if the ASN.1 file does not contain OCTET STRING and SET OF types in the module, the following definitions

```
#ifdef CODER_AUTOMATIC_CONFIG
#define CODER_NOUSE_OCTET_STRING
#define CODER_NOUSE_SET_OF
#endif
```

will be included in the configuration file.

This feature can be turned on by the `CODER_AUTOMATIC_CONFIG` compile switch. For more information about available compile switches for the configuration see "Encoding configuration" on page 2894 in chapter 58, *ASN.1 Encoding and De-coding in the SDL Suite*.

PostMaster Interface

The ASN.1 Utilities can also be invoked via the PostMaster. An example of this is when an SDL system that uses ASN.1 modules is analyzed. The Analyzer will then order the ASN.1 Utilities, via the PostMaster, to perform a translation of the ASN.1 modules to SDL packages. For a complete description of the PostMaster, see chapter 10, *The PostMaster*.

On UNIX, the PostMaster communication may also be invoked by starting `asn1util` with the `-post` command-line option. `asn1util` will then wait for commands sent to it from the PostMaster.

Translation of ASN.1 to SDL

This section describes the detailed translation rules from ASN.1 to SDL that are implemented in the ASN.1 Utilities. The translation rules all conform to Z.105, except for the cases described in "Restrictions to Z.105" on page 15 in chapter 1, *Compatibility Notes, in the Release Guide*.

General

- Case sensitivity is according to Z.105, i.e. ASN.1 names are converted directly to SDL names. This implies that in rare cases, correct ASN.1 modules may cause name conflicts when used in SDL.

Note:

Since named numbers, named bits, and integer values are all mapped to integer synonyms, the same name should not be used more than once, because this will lead to name conflicts in SDL.

- '-' (dash) in ASN.1 names is transformed to '_' (underscore), e.g. `long-name` in ASN.1 is transformed to `long_name` in SDL.
- In accordance with Z.105, tag information is ignored in the translation to SDL.
- As SDL does not have "in-line types", one ASN.1 type may be mapped to more than one SDL type. The generated in-line types get dummy names.
- External type/value references are mapped to qualifiers. For example `A.a` is mapped to `<<package A>> a`. Also a use clause (`use A;`) is generated.

Keywords substitution

ASN.1 generators can be configured to be sensitive to a certain number of identifiers. There is a special text file named 'asn1util_kwd.txt' that contains a list of identifiers and a list of their substitution during mapping. By default this file is used to configure target languages keywords substitution. It can be edited to get another functionality or another set of keywords to be replaced.

'asn1util_kwd.txt' should contain pairs of identifiers where the first one is the identifier from original ASN.1 specification that will be replaced by the second identifier during generation. 'asn1util_kwd.txt' should conform to the following syntax:

Example 33 Configuration file syntax

```
<identifier1>   <identifier1 substitution>
<identifier2>   <identifier2 substitution>
                ...
<identifierN>   <identifierN substitution>
```

ASN.1 Utility reads the first configuration file it finds. It searches for the 'asn1util_kwd.txt' file first in the current folder, then in the home folder and finally in the installation. If a configuration file named 'asn1util_kwd.txt' is put in the home folder or in the current working

folder, it will override the default configuration file from the installation. The configuration file to be used can also be specified in the Analyze dialog or from the command line with the '-K' option (see "Command-Line Interface" on page 701).

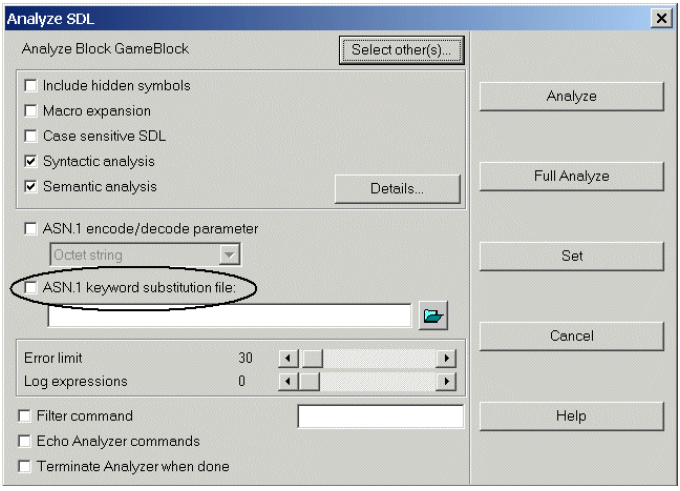


Figure 164: Keywords substitution

'asn1util_kwd.txt' is always present in the installation and it is configured to replace keywords from SDL, TTCN, C and C++ languages: the ASN.1 identifier `name`, that is a keyword in SDL, TTCN or C++, is replaced by `name_SDL_KEYWORD`, `name_TTCN_KEYWORD` or `name_CPP_KEYWORD` to avoid syntax errors in the target languages (see "Appendix A: List of recognized keywords" on page 759). If an original ASN.1 identifier has been modified, a warning message is reported. See "ERROR 2077 ASN.1 identifier #1 is a keyword, it will be replaced by #2" on page 758).

Note: Keywords recognition

ASN.1 in case-sensitive language and target language keywords are also recognized in case-sensitive mode. If generated SDL is analyzed in case-insensitive mode, there could still be keyword problems left. For example, ASN.1 contains the type named `Start` and it will not be recognized to be an SDL keyword, because the keyword `start` will be compared, but in case-insensitive mode `Start` is still a keyword in SDL, which will result in syntax errors.

Example 34 Keywords default substitution ––––––––––––––––––––

For these ASN.1 definitions:

```
CASE ::= ENUMERATED { upper, lower }

T ::= SEQUENCE {
        int INTEGER,
        explicit BOOLEAN,
        case     CASE,
        signal   INTEGER
}

value1 T ::= {
        int 5,
        explicit TRUE,
        case lower,
        signal 27 }
```

With default keywords substitution file the following SDL is generated:

```
newtype CASE_TTCN_KEYWORD
  literals upper,lower
  operators
    ordering;
endnewtype;

newtype T struct
  int_CPP_KEYWORD  Integer;
  explicit_CPP_KEYWORD  Boolean;
  case_CPP_KEYWORD  CASE_TTCN_KEYWORD;
  signal_SDL_KEYWORD  Integer;
endnewtype;

synonym value1 T = (. 5, true, lower, 27 .);
```

––

A configuration file allows the user to control the set of keywords to be replaced. Removing lines with TTCN keywords, for example, will

switch off TTCN keywords sensitivity. Providing an empty configuration file will result in switching off keywords substitution completely.

Module

- An ASN.1 module is translated to an SDL package as specified in Z.105. The `DefinitiveIdentifier` (object identifier after module name) is ignored. The tag default is also ignored.
- `EXPORTS` is mapped to a corresponding `interface`-clause.
- `IMPORTS` is mapped to a corresponding package reference clause. The `AssignedIdentifier` (object identifier after module name) is ignored.

> **Note:**
>
> **On UNIX**, the `-b` option disables generation of package/endpackage, interface and use clauses. Files that have been generated this way can be included in SDL with the #INCLUDE directive, see *"Including PR Files" on page 2506 in chapter 54, The SDL Analyzer*.

Example 35 ----------

```
MyModule DEFINITIONS ::= BEGIN

EXPORTS A, b, C;
IMPORTS X, Y, z FROM SomeModule { iso 3 0 8 }
...
END
```

is mapped to

```
package MyModule;
interface newtype A, synonym b, newtype C;
use SomeModule / newtype X, newtype Y, synonym z;
...
endpackage;
```

Joining modules

Mapping for ASN.1 original module structure can be changed by applying joining module functionality. Several ASN.1 modules can be generated into one SDL package, if ASN.1 modules are arranged into groups in the Organizer. Joining modules can also be controlled from the command line (see "Command-Line Interface" on page 701).

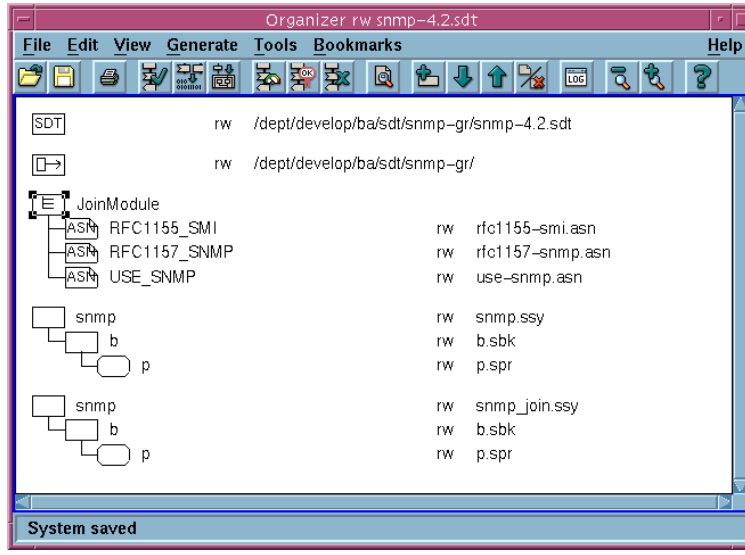


Figure 165: Joining modules

Joining means throwing away all import/export clauses and module headers and generating all module bodies into one big package with the name specified in the Organizer or command line interface.

When joining definitions from several ASN.1 modules into one SDL package, names in the resulting SDL package change according to the following rules:

- Clashed names are prefixed by the original ASN.1 module name
- Package names in the external references are replaced by the join package name

Example 36 ----------

```
M1
DEFINITIONS ::=
BEGIN

IMPORTS S2 FROM M2;

T ::= SET OF SEQUENCE { a S2 }
S1 ::= IA5String
```

```
END

M2
DEFINITIONS ::=
BEGIN

T ::= SEQUENCE OF SEQUENCE OF M1.S1
S2 ::= BOOLEAN

END
```

without joining applied ASN.1 modules M1 and M2 are mapped to

```
use M2/
  newtype S2;
package M1; /*#ASN.1 'M1'*/

newtype T
   Bag(T_INLINE_0)
endnewtype;

newtype T_INLINE_0 /*#SYNT*/ struct
  a  S2;
endnewtype;

syntype S1 = IA5String endsyntype;

endpackage M1;

package M2; /*#ASN.1 'M2'*/

newtype T
   String (T_INLINE_0, emptystring)
endnewtype;

newtype T_INLINE_0 /*#SYNT*/
   String (<<package M1>>S1, emptystring)
endnewtype;

syntype S2 = Boolean endsyntype;

endpackage M2;
```

with joining to package Join-Package applied ASN.1 modules are mapped to

```
package Join_Package; /*#ASN.1 'Join_Package'*/

newtype M1_T
  Bag(M1_T_INLINE_0)
endnewtype;
newtype M1_T_INLINE_0 /*#SYNT*/ struct
```

```
  a  S2;
endnewtype;
syntype S1 = IA5String endsyntype;

newtype M2_T
   String (M2_T_INLINE_0, emptystring)
endnewtype;
newtype M2_T_INLINE_0 /*#SYNT*/
   String (<<package Join_Package>>S1, emptystring)
endnewtype;
syntype S2 = Boolean endsyntype;

endpackage Join_Package;
```

General Type and Value Assignment

A type assignment is mapped to a newtype or a syntype, depending on the type on the right-hand side of the ‘::=’. Tags are ignored. An ASN.1 value assignment is mapped to a synonym.

Example 37

```
T1 ::= INTEGER
T2 ::= [APPLICATION 28] T1
a BOOLEAN ::= TRUE
```

is mapped to

```
syntype T1 = Integer endsyntype;
syntype T2 = T1 endsyntype;
synonym a Boolean = True;
```

Inline types naming

The ASN.1 language can use type definitions inside composite types, which are called inline types. Inline types are not allowed in SDL. In SDL, only named types can be used in a composite type. Implicit names are assigned to ASN.1 inline types and they are referenced by this name in SDL.

Implicit names for generated SDL have the following syntax: <parent_definition_name>_INLINE_<counter>, where parent_definition_name is either the name of the parent type or the name of the parent value, depending on if inline type exists in type or value assignment construct in ASN.1.

Example 38 ----------

```
T1 ::= SEQUENCE {
        a SET OF INTEGER,
        b CHOICE { x BIT STRING,
                   y OCTET STRING },
        c ENUMERATED { sat, sun } }

For type T the following inline types will be
generated to SDL

newtype T1 struct
  a  T1_INLINE_0;
  b  T1_INLINE_1;
  c  T1_INLINE_2;
endnewtype;

newtype T1_INLINE_0 /*#SYNT*/
   Bag(Integer)
endnewtype;

newtype T1_INLINE_1 /*#SYNT*/ choice
  x  Bit_string;
  y  Octet_string;
endnewtype;

newtype T1_INLINE_2 /*#SYNT*/
  literals sat,sun
  operators
    first: T1_INLINE_2 -> T1_INLINE_2;
    last:  T1_INLINE_2 -> T1_INLINE_2;
    succ:  T1_INLINE_2 -> T1_INLINE_2;
    pred:  T1_INLINE_2 -> T1_INLINE_2;
    num:   T1_INLINE_2 -> Integer;
    ordering;

  <operator definitions>

endnewtype;
```

Example 39 ----------

```
T2 SEQUENCE OF INTEGER ::= { {1,1} | {2,2} }

For T2 the following inline types will be generated
to SDL

newtype T2 /*#SYNT*/
   String (Integer, emptystring)
   constants ((. 1, 1 .)), ((. 2, 2 .))
endnewtype;
```

Example 40

```
val BIT STRING ( SIZE(3) ) ::= '101'B

synonym val val_INLINE_0 = bitstr('101');

syntype val_INLINE_0 = Bit_string constants size (3)
endsyntype;
```

Note:

SDL inline names can change if you change within the parent type or value in the ASN.1 specification, the counter can differ. If these names are used within an SDL system, then you must update the SDL system.

Boolean, NULL, and Real

`BOOLEAN`, `NULL` and REAL are mapped to the corresponding SDL types. Value notations for these types are mapped as follows

ASN.1 type	ASN.1 value	Corresponding SDL value
ANY		not supported (conform Z.105)
BOOLEAN	TRUE FALSE	True False
NULL	NULL	NULL
REAL	0 PLUS-INFINITY MINUS-INFINITY { mantissa 31416, base 10, exponent -4 }	0.0 PLUS_INFINITY MINUS_INFINITY 3.1416

If a REAL value has an exponent bigger than 1000 and if the mantissa is not zero, then it is mapped to PLUS_INFINITY or MINUS_INFINITY. If a REAL value has an exponent less than -1000, then it is mapped to 0.

Bit String

BIT STRING is mapped to the Z.105-specific type Bit_string. Named bits are mapped to integer synonyms. Values for bit strings are mapped to hexstr/bitstr expressions.

Example 41 ----------

```
B ::= BIT STRING { bit0(0), bit23(23) }

b1 BIT STRING ::= '011 1110'B
b2 BIT STRING ::= '3AFC'H
```

is mapped to

```
syntype B = Bit_string endsyntype;
synonym bit0 Integer = 0;
synonym bit23 Integer = 23;

synonym b1 Bit_string = bitstr('0110 1110');
synonym b2 Bit_string = hexstr('3AFC');
```

Note:

Bit_string, as opposed to most other string types in SDL, has indices starting with 0!

Type Bit is a Z.105 specific type with literals 0 and 1, and with boolean operators.

Available operators:

```
bitstr    : Charstring             -> Bit_string;
     /* converts a Charstring consisting of '0' and
        '1'-s to a Bit_string */
hexstr    : Charstring             -> bit_string;
     /* converts a Charstring consisting of
        hexadecimal characters to a bit_string */
"not"     : Bit_string             -> Bit_string;
"and"     : Bit_string, Bit_string -> Bit_string;
"or"      : Bit_string, Bit_string -> Bit_string;
"xor"     : Bit_string, Bit_string -> Bit_string;
"=>"      : Bit_string, Bit_string -> Bit_string;
     /* bitwise logical operators */
mkstring  : Bit                    -> Bit_string;
length    : Bit_string             -> Integer;
first     : Bit_string             -> Bit;
last      : Bit_string             -> Bit;
"//"      : Bit_string, Bit_string -> Bit_string;
extract   : Bit_string, Integer    -> Bit;
modify!   : Bit_string, Integer, Bit-> Bit_string;
substring : Bit_string, Integer, Integer ->
```

```
                                                Bit_string;
        /* normal String operators, except that index
           starts with 0;
           see also "Sequence of Types" on page 720 */
```

Character Strings

`PrintableString`, `NumericString`, `VisibleString`, and `IA5String` (i.e. all ASN.1 character string types with character sets that are a subset of ASCII) are mapped to syntypes of SDL `Charstring`. Values for these strings are mapped to corresponding `Charstring` synonyms in SDL.

The same operators as for `Charstring` are available for these types, and values of these types can be assigned freely to each other without need for conversion operators.

For example, in SDL an `IA5String` value can be assigned to a `NumericString` variable (given that the `IA5String` only contains numeric characters).

Choice Types

A `CHOICE` type is mapped to the choice-construct that is described in more detail in "Choice" on page 2670 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*.

Example 42 ----------

```
C ::= CHOICE {
        a INTEGER,
        b BOOLEAN}

c C ::= a:7
```

is mapped to

```
newtype C choice
   a Integer;
   b Boolean;
endnewtype;

synonym c C = a:7
```

The operators that are available for a `CHOICE` type are (assuming that `C` is defined as in Example 42 above):

```
aextract!       : C -> Integer;
```

```
     /* e.g. c!a returns 7 */
bextract!       : C -> Boolean;
     /* but c!b gives dynamic error! */
amake!          : Integer -> C;
bmake!          : Boolean -> C;
     /* build choice value, e.g. in SDL
        it is possible to write b:True */
amodify!        : C, Integer -> C;
     /* e.g. var!a := -5 */
bmodify!        : C, Boolean -> C;
presentextract! : C -> xxx;
     /* returns the selected field.
        xxx is an anonymous type with values a and
b.
        E.g. c!present gives a */
```

Enumerated Types

An ENUMERATED type is mapped to a newtype with a set of literals plus some operators. By default only ordering operators are generated, use command line option -e to get the rest. The list of literals that is generated is reordered in accordance with the associated integer values.

Example 43 ----------

```
N ::= ENUMERATED { yellow(5), red(0), blue(6) }
```

is mapped to (only signature of operators shown)

```
newtype N
  literals red, yellow, blue
  /* note that the literals have been reordered! */
operators
   ordering;
   first: N -> N;
   last: N -> N;
   succ: N -> N;
   pred: N -> N;
   num: N -> Integer;
 endnewtype,
```

The operators that are available for an ENUMERATED type are (assuming that N is defined as in Example 43 above):

```
num : N -> Integer;
     /* num(yellow)=5, num(red)=0, num(blue)=6 */
"<" : N, N -> Boolean;
"<=": N, N -> Boolean;
">" : N, N -> Boolean;
">=": N, N -> Boolean;
    /* comparison based on num, i.e. red < yellow */
```

```
pred: N -> N;
succ: N -> N;
     /* predecessor/successor based on num, i.e.
        succ(red)=yellow, succ(yellow)=blue,
        pred(red) gives a dynamic error */
first: N -> N;
last : N -> N;
     /* first/last element based on num, i.e.
        first(red)=red, last(red)=blue */
```

Integer

`INTEGER` is mapped to the SDL `Integer` type, and ASN.1 integer values are mapped to corresponding SDL values.

Named numbers are mapped to synonyms.

Example 44 ----------

```
A ::= INTEGER { a(5), b(7) }
```

is mapped to

```
syntype A = Integer endsyntype;
synonym a Integer = 5;
synonym b Integer = 7;
```

Object Identifier

`OBJECT IDENTIFIER` is mapped to the Z.105-specific type `Object_Identifier`. The normal String operators are available for `Object_Identifier`, listed also in "Sequence of Types" on page 720. Indices start as usual with 1.

Octet String

OCTET STRING is mapped to the Z.105-specific type Octet_string. Octet_string is based on type Octet. This type is further described in "SDL Predefined Types" on page 2658 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*. The mapping for the octet string value notation to SDL is identical to bit strings, see "Bit String" on page 714.

> **Note:**
>
> Octet_string, has indices starting with 1.

Operators available:

```
bitstr      : Charstring       -> Octet_string;
hexstr      : Charstring       -> Octet_string;
     /* conversion from Charstring to Octet_string,
        see also "Bit String" on page 714*/
bit_string  : Octet_string     -> Bit_string;
octet_string: Bit_string       -> Octet_string;
     /* conversion operators
        Octet_string <-> Bit_string */
mkstring    : Octet            -> Octet_string;
length      : Octet_string     -> Integer;
first       : Octet_string     -> Octet;
last        : Octet_string     -> Octet;
"//"        : Octet_string, Octet_string ->
                                    Octet_string;
extract!    : Octet_string, Integer -> Octet;
modify!     : Octet_string, Integer, Octet ->
                                    Octet_string;
substring   : Octet_string, Integer, Integer ->
                                    Octet_string;
     /* normal String operators, see also
        "Sequence of Types" on page 720 */
```

Sequence/Set Types

SEQUENCE and SET are both mapped to SDL *struct*. From an SDL point of view there is no difference between SEQUENCE and SET. In order to support optional and default components, SDL has been extended with corresponding concepts.

> **Note:**
>
> Optional and default fields in *struct* are both non-standardized extensions to SDL.

Values are mapped to the "(. ... .)" construct (= `Make!` operator). Values for optional and default components are not supported. Instead, SDL tasks should be used to assign optional and default components.

Example 45 ----------

```
S ::= SEQUENCE {
        a INTEGER OPTIONAL,
        b BOOLEAN,
        c IA5String DEFAULT "xyz" }

s S ::= { b TRUE }
```

is mapped to

```
newtype S struct
   a Integer optional;
   b Boolean;
   c IA5String := 'xyz';
endnewtype;

synonym s S = (. True .);
```

The operators that are available for a `SEQUENCE` or `SET` type are (assuming that `S` is defined as in Example 45 above):

```
make!    : Boolean -> S;
     /* builds a value for S */
aextract!: S -> Integer;
bextract!: S -> Boolean;
cextract!: S -> IA5String;
   /* Extract operators. Note that aextract! gives
      dynamic error if the field has not been set */
amodify! : S, Integer -> S;
bmodify! : S, Boolean -> S;
cmodify! : S, IA5String -> S;
     /* Modify operators change one component
        in a Sequence/Set */
apresent : S -> Boolean;
     /* gives True if component a has been assigned
        a value, e.g. aPresent(s) = False */
```

Sequence of Types

SEQUENCE OF is mapped to the String generator. Values are mapped to corresponding synonyms.

Example 46 ----------

```
S ::= SEQUENCE OF INTEGER
s1 S ::= { 3, 2, 5 }
s2 S ::= {}
```

is mapped to

```
newtype S
   String (Integer, Emptystring)
endnewtype;

synonym s1 S = (. 3, 2, 5 .);
synonym s2 S = (. .);
```

The normal String operators are available for Sequence types. Indices start at 1.

The operators that are available for a SEQUENCE OF type are (assuming that S is defined as in Example 46 above):

```
mkstring  : Integer             -> S;
     /* make a sequence of one item */
length    : S                   -> Integer;
     /* returns number of elements in sequence */
first     : S                   -> Integer;
     /* returns first element in sequence */
last      : S                   -> Integer;
     /* returns last element in sequence
"//"      : S, S                -> S;
     /* returns concatenation of two sequences */
extract!  : S, Integer          -> Integer;
     /* returns the indexed element */
modify!   : S, Integer, Integer -> S;
     /* modify the indexed element */
substring : S, Integer, Integer -> S;
     /* Substring(S, i, l) returns substring of S
        of length l, starting at index i */
make!     : * Integer           -> S;
      /* adds the included elements to the string,
       * corresponds to (. .) */
append    : in/out S, Integer;
      /* appends one element to the string */
```

Set of Types

`SET OF` is mapped to the Z.105 specific `Bag` generator. For a more complete description of the Bag generator, see *"Bag" on page 2673 in chapter 56, The Cadvanced/Cbasic SDL to C Compiler*.

Example 47 ----------

```
S ::= SET OF INTEGER
s1 S ::= { 2, 2, 5 }
s2 S ::= {}
```

is mapped to

```
newtype S
   Bag (Integer)
endnewtype;

synonym s1 S = (. 2, 2, 5 .);
synonym s2 S = (. .);
```

The operators that are available for a `SET OF` type are (assuming that `S` is defined as in Example 47 above):

```
incl     : Integer, S    -> S;
     /* add an element to the bag */
del      : Integer, S    -> S;
     /* delete one element */
incl     : Integer, in/out S;
del      : Integer, in/out S;
length   : S              -> Integer;
     /* returns number of elements */
take     : S              -> Integer;
     /* return some element from the bag */
take     : S,   Integer   -> Integer;
     /* return the indexed element in the bag */
makebag  : Integer        -> S;
     /* build a bag of one element */
"in"     : Integer, S     -> Boolean;
     /* gives true if the element is in the bag */
"<"      : S, S           -> Boolean;
">"      : S, S           -> Boolean;
"<="     : S, S           -> Boolean;
">="     : S, S           -> Boolean;
     /* subset/superset comparison operators */
"and"    : S, S           -> S;
"or"     : S, S           -> S;
     /* intersection/union operators */
make!    : * Integer      -> S;
     /* adds the included elements to the bag,
      * corresponds to (. .) */
```

Useful Types

The types `GeneralizedTime` and `UTCTime` have been defined in terms of ASN.1 as specified in X.680. It follows from their definition in X.680, together with the information about the translation rules given in this chapter, which operators are available in SDL for these types.

Constrained Types

Constrained types are mapped to sdl syntypes of the associated parent sort. Value constraints are mapped to sdl range condition.

When specifying ASN.1 value constraints, several constructs can be used that are not supported in the SDL Suite, such as ALL EXCEPT,INCLUDES <subtype> and value range with MIN or MAX endpoint. Possible values for such a type are computed and mapped to syntype with range condition represented by a sequence of open and closed ranges.

Example 48 ----------

```
T  ::= INTEGER ( (1..10) EXCEPT 8 )

T1 ::= INTEGER ( INCLUDES T EXCEPT (3..<6) )
```

is mapped to

```
syntype T = Integer
            constants 9 : 10, 1 : 7
endsyntype;

syntype T1 = Integer
             constants 9 : 10, 6 : 7, 1 : 2
endsyntype;
```

“COMPONENTS OF” and “WITH COMPONENT” constraints are mapped by using extra inline types. If the present constraint is applied to the parent type, then the new type is generated excluding fields marked as ABSENT and including fields marked as PRESENT.

Example 49 ----------

```
T  ::= SEQUENCE {
         a INTEGER,
         b IA5String
}

T1 ::=  T ( WITH COMPONENTS {
              a (-5..5),
```

```
                b (SIZE (7))
} )
```

is mapped to

```
newtype T struct
    a  Integer;
    b  IA5String;
endnewtype;

newtype T1 struct
    a  T1_INLINE_0;
    b  T1_INLINE_1;
endnewtype;

syntype T1_INLINE_0 = Integer constants -5 : 5
endsyntype;

syntype T1_INLINE_1 = IA5String constants size(7)
endsyntype;
```

Example 50 ----------

```
T  ::= SET OF BIT STRING

T1 ::= T ( WITH COMPONENT (SIZE (5)) )
```

is mapped to

```
newtype T
    Bag(Bit_string)
endnewtype;

syntype T1 = T1_INLINE_0 endsyntype;

newtype T1_INLINE_0 /*#SYNT*/
    Bag(T1_INLINE_1)
endnewtype;

syntype T1_INLINE_1 = Bit_string
    constants size (5)
endsyntype;
```

Example 51 ----------

```
T ::= SET {
       a INTEGER OPTIONAL,
       b REAL OPTIONAL
}
```

```
T1 ::= T ( WITH COMPONENTS {
             a (0..<MAX) PRESENT,
             b ABSENT
} )
```

is mapped to

```
newtype T struct
   a  Integer  optional;
   b  Real  optional;
endnewtype;

newtype T1 struct
    a  T1_INLINE_0;
endnewtype;

syntype T1_INLINE_0 = Integer constants >=0
endsyntype;
```

Note:

According to ASN.1, the types T and T1 are compatible, because they are derived from each other. In SDL these are different types and values of type T can not be assigned to type T1.

ASN.1 SET OF and SEQUENCE OF types with SIZE or single value constraints are mapped to one SDL type with constraint without introducing any extra inline types.

Example 52 ----------

```
T1 ::= SEQUENCE SIZE (5..15) OF INTEGER

T2 ::= SEQUENCE ( { 1 } | {} ) OF INTEGER

T3 ::= SET (SIZE (MIN .. <100) ) OF BOOLEAN

T4 ::= SET (SIZE (15) | { ''B } ) OF BIT STRING
```

is mapped to

```
newtype T1
   String(Integer, emptystring)
   constants size (5 : 15)
endnewtype;

newtype T2
   String(Integer, emptystring)
   constants ((. .)), ((. 1 .))
endnewtype;

newtype T3
```

```
   Bag(Boolean)
   constants size (<=99)
endnewtype;

newtype T4
   Bag(Bit_string)
   constants ((. bitstr('') .)), size (15)
endnewtype;
```

Extensibility

Extensibility was introduced in X.680 (1997). In ASN.1 extensibility is represented with extension markers and extension addition groups, that can be specified inside SET, SEQUENCE, CHOICE, ENUMERATED types and constraints.

Extension markers are not visible in SDL translations. All square brackets are ignored and all components from extension addition groups are translated into SDL as individual fields. All required components from extension additions, individual or from extension addition groups are mapped to optional ones.

Example 53

```
S1 ::= SET
{
  x [100] INTEGER,
  ... ,
  [[
     gr11 REAL
  ]],
  t BIT STRING,
  [[
     gr21 BOOLEAN OPTIONAL,
     gr22 SET OF INTEGER
  ]],
  ... ,
  y INTEGER
}
```

is mapped to SDL

```
newtype S1 struct
  x  Integer;
  gr11  Real  optional;
  t  Bit_string optional;
  gr21  Boolean  optional;
  gr22  S1_INLINE_1  optional;
  y  Integer;
```

```
endnewtype;

newtype S1_INLINE_1 /*#SYNT*/
   Bag(Integer)
endnewtype;
```

Note:

SDL translation removes the borders of additional groups and makes all required components optional.The semantics for assigning values to types with additional groups are: either the whole addition group ([[....]]) is absent, or it is all present unless components inside the group are optional. This is not checked in SDL tools but inconsistency will cause errors in ASN.1 encoding.

Extension markers are ignored in constraints. If both root and additional constraints are present, they are translated to the union constraint.

Example 54

```
T1 ::= INTEGER ( 1..10 ^ 2..20, ... , 12 )

is mapped to SDL

syntype T1 = Integer constants 12, 2 : 10
endsyntype;

T2 ::= INTEGER (1 | 3, ... )

is mapped to SDL

syntype T2 = Integer constants 3, 1 endsyntype;
```

Information from Object Classes, Objects and Object Sets

Object classes, object and objects sets are not translated to SDL. Only types and values are translated to SDL, but it is possible in ASN.1 to use information from object classes, objects and object sets when specifying types and values. This information is translated into SDL.

ObjectClassFieldType

ObjectClassFieldType is a reference to object class and a field in that class. The translation to SDL depends on the kind of field name used.

An open type is defined if the field name references a type field, a variable type value field or variable type value set field. An open type can be any ASN.1 type. Open types are translated to Octet_string types in SDL.

Example 55 ----------

```
OPERATION ::= CLASS {
   &ArgumentType,
   &arg &ArgumentType
}

T1 ::= SEQUENCE { a OPERATION.&ArgumentType }
```

is translated to SDL

```
newtype T1 struct
  a  T1_INLINE_0;
endnewtype;

syntype T1_INLINE_0 = Octet_string endsyntype;


T2 ::= OPERATION.&arg
```

is translated to SDL

```
syntype T2 = Octet_string endsyntype;
```

If the field name in the class references a fixed type value or fixed type value set fields, then the fixed type is used when translated to SDL.

Example 56 ----------

```
OPERATION ::= CLASS {
      &ValueSet INTEGER
   }

T ::= OPERATION.&ValueSet
```

is translated to SDL

```
syntype T = Integer endsyntype;
```

ObjectClassFieldType with table constraint (object set constraint)

Table constraint applied to ObjectClassFieldType restricts the set of possible types or values to those specified in a column of the table. A table corresponds to an object set. The columns of the table correspond to the object class fields and the rows correspond to the objects in the set.

If the field name in ObjectClassFieldType is a type field and constrained with a table, then it is translated to a CHOICE type with fields of the types specified in the table column. The names of the fields in the choice are the same as the names of the types in the column but the first letter is changed from upper case to lower case.

Note: Field names

If the type in the field is inline then the name in the field will be an implicitly generated inline name, like t_INLINE_4.

If the field name in ObjectClassFieldType is a fixed type value or a fixed type value set, then this is translated to a constrained type where only values that are specified in the table column are permitted.

If the field name in ObjectClassFieldType is a variable type value or variable type value set field, then this is translated to a CHOICE type with types, that are constrained to have values specified in the corresponding cell in the same row of the table.

Example 57 ––

```
OPERATION ::= CLASS {
       &ArgumentType,
       &operationCode INTEGER UNIQUE,
       &ValueSet INTEGER,
       &ArgSet &ArgumentType
   }
```

The My-Operations object set

Object name	&ArgumentType	&operationCode	&ValueSet	&ArgSet
operationA	INTEGER	1	{1 | 2 | 5 .. 8}	{ 111..444 }

Object name	&ArgumentType	&operationCode	&ValueSet	&ArgSet
operationB	SET OF INTEGER	2	{ 2 .. 8 }	{ {1,2,3} | { 888 } }

```
C1 ::= OPERATION.&ArgumentType ( {My-Operations} )
```

is translated to SDL

```
newtype C1 choice
  integer  Integer;
  c1_INLINE_2  C1_INLINE_1;
endnewtype;

newtype C1_INLINE_1 /*#SYNT*/
   Bag(Integer)
endnewtype;
```

```
C2 ::= OPERATION.&operationCode ( {My-Operations} )
```

is translated to SDL

```
syntype C2 = Integer constants 2, 1 endsyntype;
```

```
C3 ::= OPERATION.&ValueSet ( {My-Operations} )
```

is translated to SDL

```
syntype C3 = Integer constants 2 : 8, 1, 5 : 8, 2
endsyntype;
```

```
C4 ::= OPERATION.&ArgSet ( {My-Operations} )
```

is translated to SDL

```
newtype C4 choice
  c4_INLINE_1  C4_INLINE_1;
  c4_INLINE_1  C4_INLINE_2;
endnewtype;

syntype C4_INLINE_1 = Integer constants 111 : 444
endsyntype;

syntype C4_INLINE_2 /*#SYNT*/
  Bag(Integer)
  constants ((. 888 .)), ((. 1, 2, 3 .))
endsyntype;
```

If an open type is constrained by the table for which all type settings are omitted, then it is translated to SDL Octet_string instead of an empty CHOICE type.

Example 58 ----------

```
MY-CLASS ::= CLASS {
       &id INTEGER,
       &OpenType OPTIONAL
   }
```

The My-Set object set:

Object name	&id	&OpenType
object1	1	-
object2	2	-

```
S ::= SEQUENCE {
        id  MY-CLASS.&id({My-Set}),
        val MY-CLASS.&OpenType({My-Set}{@id})
}
```

is translated to SDL

```
newtype S struct
  id  S_INLINE_0;
  val  S_INLINE_2;
endnewtype;

syntype S_INLINE_0 = Integer constants 2, 1
endsyntype;

syntype S_INLINE_2 = Octet_string endsyntype;
```

TypeFromObject

TypeFromObject is a reference to an object and a type field in that object. This is simply translated to that type in SDL. If the field is optional in the class and not set in the object, then TypeFromObject cannot be translated.

Example 59 ----------

```
OPERATION ::= CLASS {
       &ArgumentType,
```

```
        &ResultType
    }

operationA OPERATION ::= {
      &ArgumentType INTEGER,
       &ResultType BOOLEAN
    }

O1 ::= operationA.&ArgumentType
```

is translated to SDL

```
syntype O1 = Integer endsyntype;


O2 ::= operationA.&ResultType
```

is translated to SDL

```
syntype O2 = Boolean endsyntype;
```

ValueSetFromObject

ValueSetFromObject is a reference to an object and a field with a set of values in that object. This is translated to a constrained type in SDL, allowing only values from the value set.

Example 60 ----------

```
OPERATION ::= CLASS {
        &ValueSet INTEGER
    }

operationA OPERATION ::= {
        &ValueSet { 1 | 2 | 5..8 }
  }

V1 ::= operationA.&ValueSet
```

is translated to SDL

```
syntype V1 = Integer constants 2, 5 : 8, 1
endsyntype;
```

ValueFromObject

ValueFromObject is a reference to an object and a field with a value in that object. This is translated to the same value in SDL.

Example 61 ----------

```
OPERATION ::= CLASS {
      &operationCode INTEGER UNIQUE
   }

operationA OPERATION ::= { &operationCode 1 }

val2 INTEGER ::= operationA.&operationCode
```

is mapped to SDL

```
synonym val2 Integer = 1;
```

CONSTRAINED BY notation

CONSTRAINED BY is treated like a comment and is not translated to SDL.

Parameterization

Wherever a parameterized type or value is used, it is translated to SDL after all dummy references are replaced by the actual parameters. A parameterized value is also translated after all dummy references are replaced by the actual parameters.

Parameterized assignments are ignored when translating to SDL.

Example 62 ----------

```
Container { ElemType, INTEGER : maxelements } ::=
       SET SIZE (0..maxelements ) OF ElemType

Intcontainer ::= Container {INTEGER, 25}
```

is first internally mapped to

```
Intcontainer ::= SET SIZE( 0..25 ) OF INTEGER
```

and then translated to SDL. Container is not translated to SDL.

Support for External ASN.1 in the TTCN Suite

The ASN.1 Utilities are also used by the TTCN Suite if a TTCN test suite contains data types and constraints that are defined in the tables “ASN.1 Type Definitions By Reference” and “ASN.1 Constraints By Reference”. For more information, see “ASN.1 External Type/Value References” on page 1197 in chapter 26, *Analyzing TTCN Documents (on UNIX)*.

Since TTCN is based on the older X.228 standard, while the ASN.1 Utilities are based on the new X.680 standard, users should be careful to use the common subset of X.680 and X.228 if an ASN.1 module is to be used in TTCN. In particular there are a number of differences:

- In `ENUMERATED` types, a value must be supplied for all values. For example:

  ```
  E ::= ENUMERATED { a, b }
  ```

  should be replaced by

  ```
  E ::= ENUMERATED { a(0), b(1) }
  ```

- X.680 offers more possibilities for specifying constraints than X.228 does. X.228 does not have the keywords `ALL`, `EXCEPT`, `UNION`, and `INTERSECTION`.

- For ASN.1 types that have components (e.g. `SET` or `SEQUENCE`), an identifier must be provided for every component (according to X.680), while in X.228 identifiers can be omitted. For example:

  ```
  S ::= SEQUENCE { INTEGER } -- valid X.228
  ```

  This is invalid according to X.680. The following should be used instead:

  ```
  S ::= SEQUENCE { field1 INTEGER }
  ```

General

- ‘-’ (dash) in ASN.1 names is transformed to ‘_’ (underscore), e.g. `long-name` in ASN.1 is transformed to `long_name` in TTCN.

- In general ASN.1 to TTCN translation look like pretty printing of ASN.1 modules into TTCN tables for most of the constructs, but not

all of them. Some ASN.1 concepts are not supported in TTCN Suite, they have to be modified during TTCN generation:

- Concepts defined in X.681, X.682 and X.683 (see "Information from Object Classes, Objects and Object Sets" on page 740, "CONSTRAINED BY notation" on page 745 and "Parameterization" on page 745)
- automatic tagging (see "Keywords substitution" on page 734)
- COMPONENTS OF Type notation (see "COMPONENTS OF Type notation" on page 737)
- selection types (see "Selection types" on page 738)
- enumerated types without numbers for enum identifiers (see "Enumerated types" on page 738)
- extensibility (see "Extensibility" on page 725)

Keywords substitution

ASN.1 generators can be configured to be sensitive to a certain number of identifiers. There is a special text file named 'asn1util_kwd.txt' that contains a list of identifiers and a list of their substitution during mapping. By default this file is used to configure target languages keywords substitution. It can be edited to get another functionality or another set of keywords to be replaced.

'asn1util_kwd.txt' should contain pairs of identifiers where the first one is the identifier from the original ASN.1 specification that will be replaced by the second identifier during generation. 'asn1util_kwd.txt' should conform to the following syntax:

Example 63 Configuration file syntax ----------

```
<identifier1>   <identifier1 substitution>
<identifier2>   <identifier2 substitution>
                ...
<identifierN>   <identifierN substitution>
```

ASN.1 Utility reads the first configuration file it finds. It searches for 'asn1util_kwd.txt' file first in the current folder, then in the home folder and finally in the installation. If a configuration file named 'asn1util_kwd.txt' is put in the home folder or in the current working

folder, it will override the default configuration file from the installation. The configuration file to be used can also be specified in the command line with the '-K' option (see "Command-Line Interface" on page 701), for example,

Example 64 Configuration file specification ––––––––––––––––––––

```
asn1util -K my_config.txt -i File.ttcn File.asn
```

––

'asn1util_kwd.txt' is always present in the installation and it is configured to replace keywords from the SDL, TTCN, C and C++ languages: the ASN.1 identifier `name`, that is a keyword in SDL, TTCN or C++, is replaced by `name_SDL_KEYWORD`, `name_TTCN_KEYWORD` or `name_CPP_KEYWORD` to avoid syntax errors in the target languages (see "Appendix A: List of recognized keywords" on page 759). If an original ASN.1 identifier has been modified, a warning message is reported. See "ERROR 2077 ASN.1 identifier #1 is a keyword, it will be replaced by #2" on page 758).

Example 65 Keywords default substitution ––––––––––––––––––––

For these ASN.1 definitions:

```
CASE ::= ENUMERATED { upper, lower }

T ::= SEQUENCE {
        int INTEGER,
        explicit BOOLEAN,
        case     CASE,
        signal   INTEGER
}

value1 T ::= {
        int 5,
        explicit TRUE,
        case lower,
        signal 27 }
```

With default keywords substitution file the following TTCN is generated:

```
CASE_TTCN_KEYWORD ::=
  ENUMERATED {upper(0), lower(1)}

T ::= SEQUENCE {
        int_CPP_KEYWORD INTEGER,
        explicit_CPP_KEYWORD BOOLEAN,
```

```
        case_CPP_KEYWORD CASE_TTCN_KEYWORD,
        signal_SDL_KEYWORD INTEGER
}

value1 T ::= {
  int_CPP_KEYWORD 5,
  explicit_CPP_KEYWORD TRUE,
  case_CPP_KEYWORD lower,
  signal_SDL_KEYWORD 27
}
```

A configuration file allows user to control the set of keywords to be replaced. Removing lines with SDL keywords, for example, will switch off SDL keywords sensitivity. Providing an empty configuration file will result in switching off keywords substitution completely.

Automatic tagging

If 'AUTOMATIC TAGS' is written in the header of an external ASN.1 module, then implicit tags are inserted into SET, SEQUENCE and CHOICE types. During the TTCN generation they are inserted in the type definitions explicitly.

Example 66

```
M1
DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
T ::= SEQUENCE
{
   a INTEGER OPTIONAL,
   b INTEGER DEFAULT 5
}

C ::= CHOICE
  {
    x INTEGER,
    y BOOLEAN,
    z REAL
  }
END

is translated to TTCN

SEQUENCE
{
  a [0] INTEGER OPTIONAL,
  b [1] INTEGER DEFAULT 5
}
```

```
CHOICE
{
  x [0] INTEGER,
  y [1] BOOLEAN,
  z [2] REAL
}
```

COMPONENTS OF Type notation

COMPONENTS OF Type can appear in SET or SEQUENCE field types. Instead of COMPONENTS OF Type a list of components of the referenced type is included, except extension addition components.

Example 67

```
 S1 ::= SEQUENCE
{
  x INTEGER,
  g NULL,
  ... ,
  [[
     y BOOLEAN,
     z BIT STRING
  ]],
  [[
     c IA5String
  ]],
  d SET OF
    INTEGER OPTIONAL,
  ... ,
  f REAL
}

S2 ::= SEQUENCE
 {
   a IA5String,
   COMPONENTS OF S1,
   b OCTET STRING
 }

Type S2 is translated to TTCN

SEQUENCE
{
  a IA5String,
  x INTEGER,
  g NULL,
  f REAL,
  b OCTET STRING
}
```

Selection types

A selection type is mapped to the type it denotes.

Example 68 ----------

```
C ::= CHOICE
 {
   a INTEGER,
   b BOOLEAN
 }

T1 ::= a < C

T1 is translated to TTCN

INTEGER

T2 ::= b < C

T2 is translated to TTCN

BOOLEAN
```

Enumerated types

Enumerated items can be defined using “identifier” notation or “identifier and number” notation. For “identifier” notations, implicit numbers are assigned to the identifiers according to the rules described in X.680 (1997), 19.

For TTCN, all enumeration items are generated with their corresponding numbers using “identifier and number” notation, and they are arranged according to their number values in ascending order in the generated enumeration.

Extension markers are ignored.

Example 69 ----------

```
A ::= ENUMERATED { a, b, c(0), d, e(2) }

is translated to TTCN

ENUMERATED { c(0), a(1), e(2), b(3), d(4) }

B ::= ENUMERATED { a, b(3), ... , c(1) }

is translated to TTCN
```

```
ENUMERATED { a(0), c(1), b(3) }
```

Extensibility

Extensibility was introduced in X.680 (1997). In ASN.1 extensibility is represented with extension markers and extension addition groups, that can be specified inside SET, SEQUENCE, CHOICE, ENUMERATED types and constraints.

Extension markers are not visible in TTCN translation. All square brackets are ignored and all components from extension addition groups are translated into TTCN as individual fields. All required components from extension additions, individual or from extension addition groups, are mapped to optional ones.

Example 70

```
S1 ::= SET
{
  x [100] INTEGER,
  ... ,
  [[
     gr11 REAL
  ]],
  t BIT STRING,
  [[
     gr21 BOOLEAN OPTIONAL,
     gr22 SET OF INTEGER
  ]],
  ... ,
  y INTEGER
}

is translated to TTCN

SET
 {
   x [100] INTEGER,
   gr11 REAL OPTIONAL,
   t BIT STRING OPTIONAL,
   gr21 BOOLEAN OPTIONAL,
   gr22 SET OF INTEGER OPTIONAL,
   y INTEGER
 }
```

Note:

TTCN translation removes the borders of additional groups and makes all required components optional. The semantics for assigning values to types with additional groups is: either the whole addition group ([[....]]) is absent, or it is all present unless components inside the group are optional. This is not checked in TTCN tools but inconsistency will cause errors in ASN.1 encoding.

```
Extension markers are ignored in constraints. If
both root and additional constraints are present,
they are translated to the union constraint.
```

Example 71 ----------

```
T1 ::= INTEGER ( 1..10 ^ 2..20, ... , 12 )

is translated to TTCN

INTEGER (( 1..10 ^ 2..20 ) | ( 12 ) )

T2 ::= INTEGER (1 | 3, ... )

is translated to TTCN

INTEGER (1 | 3 )
```

Information from Object Classes, Objects and Object Sets

Object classes, object and objects sets are not translated to TTCN. Only types and values are translated to TTCN, but it is possible in ASN.1 to use information from object classes, objects and object sets when specifying types and values. This information is translated into TTCN.

ObjectClassFieldType

ObjectClassFieldType is a reference to an object class and a field in that class. The translation to TTCN depends on the kind of field name used.

An open type is defined if the field name references a type field, a variable type value field or variable type value set field. An open type can be any ASN.1 type. Open types are translated to OCTET STRING types in TTCN.

Example 72 ----------

```
OPERATION ::= CLASS {
   &ArgumentType,
   &arg &ArgumentType
}

T1 ::= SEQUENCE { a OPERATION.&ArgumentType }
```

is translated to TTCN

```
SEQUENCE { a  OCTET STRING }

T2 ::= OPERATION.&arg
```

is translated to TTCN

```
OCTET STRING
```

If the field name in the class references a fixed type value or fixed type value set fields, then the fixed type is used when translated to TTCN.

Example 73 ----------

```
OPERATION ::= CLASS {
      &ValueSet INTEGER
   }

T ::= OPERATION.&ValueSet
```

is translated to TTCN

```
INTEGER
```

ObjectClassFieldType with table constraint (object set constraint)

A table constraint applied to ObjectClassFieldType restricts the set of possible types or values to those specified in a column of the table. A table corresponds to an object set. The columns of the table correspond to the object class fields and the rows correspond to the objects in the set.

If the field name in ObjectClassFieldType is a type field and constrained with a table, then it is translated to a CHOICE type with fields of the types specified in the table column. The names of the fields in the choice are the same as the names of the types in the column but the first letter is changed from upper case to lower case.

Note:

If the type in the field is inline then the name in the field will be an implicitly generated inline name, like t_INLINE_4.

If the field name in ObjectClassFieldType is a fixed type value or a fixed type value set, then this is translated to a constrained type where only values that are specified in the table column are permitted.

If the field name in ObjectClassFieldType is a variable type value or variable type value set field, then this is translated to a CHOICE type with types, that are constrained to have values specified in the corresponding cell in the same row of the table.

Example 74 ––

```
OPERATION ::= CLASS {
       &ArgumentType,
       &operationCode INTEGER UNIQUE,
       &ValueSet INTEGER,
       &ArgSet &ArgumentType
   }
```

The My-Operations object set:

Object name	&ArgumentType	&operationCode	&ValueSet	&ArgSet
operationA	INTEGER	1	{1 | 2 | 5 .. 8}	{ 111..444 }
operationB	SET OF INTEGER	2	{ 2 .. 8 }	{ {1,2,3} | { 888 } }

```
C1 ::= OPERATION.&ArgumentType ( {My-Operations} )
```

is translated to TTCN

```
CHOICE {
  integer INTEGER,
  c1_INLINE_2 SET OF INTEGER
}
```

```
C2 ::= OPERATION.&operationCode ( {My-Operations} )
```

is translated to TTCN

```
INTEGER ( 1 | 2 )
```

```
C3 ::= OPERATION.&ValueSet ( {My-Operations} )
```

is translated to TTCN

```
INTEGER ( ( 1 | 2 | 5..8 ) | ( 2..8 ) )
```

```
C4 ::= OPERATION.&ArgSet ( {My-Operations} )
```

is translated to TTCN

```
CHOICE {
  c4_INLINE_1 INTEGER ( (111..444) ),
  c4_INLINE_2 SET OF INTEGER ( ( {1,2,3} | {888} ) )
}
```

If an open type is constrained by the table for which all type settings are omitted, then it is translated to TTCN OCTET STRING instead of an empty CHOICE type.

Example 75 ----------

```
MY-CLASS ::= CLASS {
       &id INTEGER,
       &OpenType OPTIONAL
   }
```

The My-Set object set:

Object name	&id	&OpenType
object1	1	-
object2	2	-

```
S ::= SEQUENCE {
        id  MY-CLASS.&id({My-Set}),
        val MY-CLASS.&OpenType({My-Set}{@id})
}
```

is translated to TTCN

```
SEQUENCE {
  id    INTEGER ( 1 | 2 ),
  val   OCTET STRING
}
```

TypeFromObject

TypeFromObject is a reference to an object and a type field in that object. This is simply translated to the that type in TTCN. If the field is optional in the class and not set in the object, then TypeFromObject cannot be translated.

Example 76 ----------

```
OPERATION ::= CLASS {
       &ArgumentType,
       &ResultType
   }

operationA OPERATION ::= {
      &ArgumentType INTEGER,
       &ResultType BOOLEAN
   }

O1 ::= operationA.&ArgumentType
```

is translated to TTCN

```
INTEGER

O2 ::= operationB.&ResultType
```

is translated to TTCN

```
BOOLEAN
```

ValueSetFromObject

ValueSetFromObject is a reference to an object and a field with a set of values in that object. This is translated to a constrained type in TTCN, allowing only values from the value set.

Example 77 ----------

```
OPERATION ::= CLASS {
       &ValueSet INTEGER
   }

operationA OPERATION ::= {
       &ValueSet { 1 | 2 | 5..8 }
  }

V1 ::= operationA.&ValueSet
```

is translated to TTCN

```
INTEGER ( 1 | 2 | 5..8 )
```

ValueFromObject

ValueFromObject is a reference to an object and a field with a value in that object. This is translated to the same value in TTCN.

Example 78

```
OPERATION ::= CLASS {
      &operationCode INTEGER UNIQUE
   }

operationA OPERATION ::= { &operationCode 1 }

val2 INTEGER ::= operationA.&operationCode
```

is translated to TTCN

```
val2 of type INTEGER equal to 1
```

CONSTRAINED BY notation

CONSTRAINED BY is treated like a comment and is not translated to TTCN.

Parameterization

Wherever a parameterized type or value is used, it is translated to TTCN after all dummy references are replaced by the actual parameters. A parameterized value is also translated after all dummy references are replaced by the actual parameters.

Parameterized assignments are ignored when translating to TTCN.

Example 79

```
Container { ElemType, INTEGER : maxelements } ::=
       SET SIZE (0..maxelements ) OF ElemType

Intcontainer ::= Container {INTEGER, 25}
```

is mapped to TTCN

```
SET SIZE ( 0..25 ) OF INTEGER
```

Error and Warning Messages

This section contains a list of the error and warning messages in the ASN.1 Utilities. Each message has a short explanation and, where applicable, a reference to the appropriate section of the recommendations X.680,X.681,X.682, X.683 or Z.105.

Some messages include a reference to the object that is the source of the diagnostic. These messages adhere to the format adopted in the SDL Suite. See chapter 18, *SDT References* for a reference to this format and for examples.

WARNING 2000 Unknown option '#1'

This warning message indicates that the ASN.1 Utilities were started with an unknown option. See "Command-Line Interface" on page 701 for an overview of the valid options.

WARNING 2001 No #1 specified after '#2' option

This warning message indicates that the ASN.1 Utilities were started with an illegal combination of options. See "Command-Line Interface" on page 701 for an overview of the valid options.

For example,

```
asn1util -s -r MyModule.asn
```

In this case no output file for sdl generation is specified after '-s' option.

ERROR 2002 Too many errors

This error message indicates that the maximum number of errors was reached when analyzing an ASN.1 module. The analysis has been aborted by the ASN.1 Utilities.

ERROR 2003 Multiple #1 paths

This warning message indicates an incorrect usage of the options of the ASN.1 Utilities.

For example,

```
asn1util -Tdir1 -Tdir2 -s MyModule.pr MyModule.asn
```

Multiple target directories provide a warning message

WARNING 2004 Option missing

This warning message indicates that no option is specified after dash.

ERROR 2005 Can not open #1

This error message indicates that an error occurred when the ASN.1 Utilities attempted to open a file. Modify, if necessary, the file protection and try to run the ASN.1 Utilities again. If the error persists, contact IBM Rational Customer Support. Contact information for IBM Rational Customer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*.

For example,

```
asn1util -Tdir -i MyModule.ttcn MyModule.asn
```

This command line can cause an error message "Can not open air/MyModule.ttcn" if there is no target directory 'dir' in the catalogue from which asn1util is called.

ERROR 2006 Illegal characters in bstring

This message indicates that an ASN.1 binary string item (used in BIT STRING and OCTET STRING) contains illegal characters. The only characters allowed are `0', `1' and white space characters. (X.680: 9.9)

ERROR 2007 Illegal characters in hstring

This message indicates that an ASN.1 hexadecimal string item (used in BIT STRING and OCTET STRING) contains illegal characters. The only characters allowed are `0'-'9', `A'-'F' and white space characters. (X.680: 9.10)

For example: 'F30C 973D'H is a valid hexadecimal string item.

ERROR 2008 'H' or 'B' expected

This error message indicates that an ASN.1 BIT STRING or OCTET STRING value is not ended with a `B or an `H. (X.680: 9.9 and 9.10).

For example: '0110'B or '1AFC'H are valid values for BIT STRING and OCTET STRING, '01110' is illegal.

ERROR 2009 Unclosed #1 string

This error is reported when there is no closing apostrophe at the end of string

WARNING 2010 Unknown token `#1'

This warning indicates a syntax error in the ASN.1 module.

ERROR 2011 Syntax error

This message indicates a syntax error in the ASN.1 module with syntax from standard X.680-X.683. This could be caused by a misspelling. It could also be caused by X.228 constructs that are not part of X.680.

ERROR 2012 Out of memory

This message indicates that the ASN.1 Utilities ran out of memory. Try to make the ASN.1 module smaller or supply more memory. If the error persists, contact IBM Rational Customer Support. Contact information for IBM Rational Customer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*.

WARNING 2013 No semantic support for `#1'

This warning indicates that an ASN.1 construct is used that is not supported by the ASN.1 Utilities. The construct will be ignored by the ASN.1 Utilities.

ERROR 2014 Export-file `#1' corrupt

This message indicates that the export file format of an ASN.1 module was corrupt or unknown. This error should normally not occur. Contact IBM Rational Customer Support. Contact information for IBM Rational Customer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*.

ERROR 2015 Old ASN1,#1

This message indicates that an ASN.1 construct of the older X.228 recommendation is used that has been superseded in the X.680 Recommendation.

For example:

```
S ::= SEQUENCE { INTEGER }
```

is old ASN.1. Correct X.680 ASN.1 is:

```
S ::= SEQUENCE { field1 INTEGER }
```

ERROR 2016 Recursive expansion of COMPONENTS OF in type #1

This error message indicates that the ASN.1 type uses directly or indirectly COMPONENTS OF itself.

ERROR 2017 Recursive #1

This message indicates that the ASN.1 construct is recursively defined.

For example: `T1::=T2, T2::=T1; T::=SET OF T`, or v T ::= v

ERROR 2018 Recursive #1 constraint

This error message indicates that type being constrained is recursively used in applied constraint.

For example: `I ::= INTEGER ( 1 .. 10 | INCLUDES I)`

ERROR 2019 Field '#1' should be initialized by #2

This message is reported when you assign wrong kind of value for the field in the object, for example when you try to assign a value for the type field in the object

ERROR 2020 Value for `#1' can not be #2

This error message indicates a semantic error in the ASN.1 module.

For example, `T ::= BIT STRING { a(-1)}` causes the error “Value for 'named bit' can not be negative”

WARNING 2021 Construct `#1' has no mapping in SDL

This warning indicates that an ASN.1 construct is used that can not be mapped to SDL.

For example:

```
S1 SEQUENCE ::= { }
       -- empty SEQUENCE/SET
    s SEQUENCE { a INTEGER OPTIONAL } ::= {}
         -- value for SEQUENCE/SET without components
```

ERROR 2022 Ambiguous reference, symbol `#1' imported more than once

A value is used that is imported more than once. Use an external value reference to specify unambiguously the module of the value that you want to use.

ERROR 2023 Multiple definition of #1

This error message appears, when the same identifier appears more then once on the right side of assignment.

For example, `X ::= INTEGER, X ::= SET OF REAL`

ERROR 2024 Exported symbol #1 not defined

This error message is reported when symbol is exported, but it is neither defined in the module nor imported to it

ERROR 2025 Ambiguous export, symbol #1 is imported more than once

This error message indicates that it is impossible to decide which symbol to export, because two symbols with the name #1 are imported to the module

ERROR 2026 Ambiguous export, symbol #1 is defined and imported

This error message indicates that it is impossible to decide which symbol to export, because symbol #1 is defined in the module and imported to it at the same time

ERROR 2027 Nothing known about module #1

This message appears when module referenced from imports clause does not exist. You should specify all modules from which symbols are imported to the analyzed module in the same command line, otherwise it is impossible to perform global semantic analysis

ERROR 2028 Import from empty module #1

This message appears when importing symbols from a module, that does not contain any definitions

ERROR 2029 Module does not export symbols

This message appears when you are trying to import symbols from module with empty export: “EXPORTS ;”

ERROR 2030 Imported symbol #1 is not exported from module #2

This message appears when symbol #1 is present in imports from module #2 clause, but it is not exported from #2. `"EXPORTS ;"` indicates that nothing is exported, while empty exports clause indicates that all definitions are exported from the module.

ERROR 2031 Imported symbol #1 is not defined in module #2

This error situation occurs when symbol is imported from module that exports all, but symbol is not defined there

ERROR 2032 Ambiguous import, symbol #1 imported more than once to module #2

This indicates that all symbols are exported from module #2, but it is impossible to import symbol #1 from module #2 because symbol #1 is imported more than once to #2. The symbols have the same name, but defined in different modules.

ERROR 2033 Ambiguous import, symbol #1 defined and imported to module #2

This indicates that all symbols are exported from module #2, but it is impossible to import symbol #1 from module #2 due to ambiguity between symbol #1defined in module #2 and symbol #1 imported to module #2.

For example;

```
M1 DEFINITIONS ::= BEGIN
   IMPORTS a FROM M2;
END

M2 DEFINITIONS ::= BEGIN
   IMPORTS a FROM M3;
   a INTEGER ::=5
END

M3 DEFINITIONS ::= BEGIN
   EXPORTS a;
  a BOOLEAN ::= TRUE

END
```

In the above case you can not import a to M1, although a is exported from M2.

ERROR 2034 Multiple declaratiom of module name #2

Module name shall appear only once in IMPORTS clause.

For example

IMPORTS a , b FROM X c FROM X; is wrong ASN.1 declaration

ERROR 2035 Recursive import for #1

This error message is reported, for example, when module A imports T from B, and B imports T from A at the same time

ERROR 2036 Multiple occurance of #1 '#2' in #3

This error is reported when some types are defined incorrectly - they have the same identifier, for example enumeration can not have the same identifiers, named number list for INTEGER type can not have the same identifiers in the list, #2 is a string

ERROR 2037 Multiple occurance of #1 #2 in #3

The same class of error as ERROR 2036 above, but #2 is an integer value.

ERROR 2038 External references are not allowed

When imports clause looks like “IMPORTS ;”,no external references are allowed from the module (X.680, 10.14, d), NOTE 2)

ERROR 2039 Referenced #1 '#2' not defined

This error is reported when you use reference that is not assigned value or type anywhere.

ERROR 2040 Value of type #1 needed

This error message indicates that value does not correspond to the type. For example `x INTEGER ::= TRUE` - this results in an error “Value of type INTEGER needed”

ERROR 2041 #1 type needed after COMPONENTS OF

The type after in COMPONENTS OF expansion should be either SET or SEQUENCE, and it should be the same as the type to which it is extracted.

For example SET { a INTEGER, b COMPONENTS OF T }, where T is SEQUENCE type is wrong usage of COMPONENTS OF notation(X.680, 22.4, 24.2)

ERROR 2042 Field names in type after COMPONENT OF already declared

After performing the COMPONENTS OF transformation, all field names should be distinct.

For example, type S1 is wrong (it has two fields named 'a')

```
S ::= SET { a INTEGER, b REAL }
S1 ::= SET { a SET OF IA5String, COMPONENTS OF S }
```

ERROR 2043 #1 type needed

This error is reported when type in selection type is not choice.

For example x < INTEGER does not satisfy that requirement

ERROR 2044 No alternative named #1 in Choice type

This error is reported when type notation is “#1 < type”, type is a CHOICE type, but it does not have alternative named #1

ERROR 2045 Too many components

This error message appears when you are trying to assign extra components, which are not defined in the type, when specifying the value of SET or SEQUENCE

ERROR 2046 No such field '#1' in #2 type

This error indicates that type #2 does not have field named #1, but you are trying to assign it a value.

ERROR 2047 Missing values for non-optional #1 fields : #2

This message indicates that not all required #1 components have been initialized in the value, and #2 is the list of names of fields, for which values are missing. The example S ::= SET { a INTEGER, b REAL, c

NULL OPTIONAL }, s S ::= { a 57 } results in error message “Missing values for non-optional SET fields: ’b’”.

ERROR 2048 More than one #1 for the field '#2'

This error occurs when you are trying to assign more then one component to one field.

For example

```
T ::= SET { a IA5String , b NULL }
t T ::= { a "val1", b NULL, a "val2" }
```

ERROR 2049 Nothing known about bit named '#1'

This error is reported when bitstring value contains identifier that is not declared in the correspondent type definition

ERROR 2050 Value for #1 should be #2

If you specify table for the value of IA5String, TableColumn should be in the range from 0 to 7, if this constraint is violated then the above error message is reported

ERROR 2051 Type is required to be derived from #1

This error indicates that type used in SubtypeConstraint is not derived from the type being constrained and thus does not satisfy X.680,45.3.2

ERROR 2052 Can not apply #1 constraint to #2 type

Not all constraints can be applied to every type, X.680, Table 6 describes which constraints can be applied to which types, if the requirements declared in Table 6 are not satisfied, the above error message is reported

ERROR 2053 There shall be at most one #1

Two presents constraints are not allowed when constraining a CHOICE type.

For example:

```
C ::= CHOICE { a T1 , b T2 } (WITH COMPONENTS {a PRESENT,
b PRESENT} )
```

causes the error message

ERROR 2054 Wrong value : out of constraint

This error is reported when value does not correspond to the constraint applied to the type.

For example: `x INTEGER (1..10) ::= -1`, x is out of constraint

ERROR 2055 The same tags for #1 components

This error message indicates that type does not correspond to the requirements for distinct tags specified in X.680, 22.5, 24.3, 26.2; If you use AUTOMATIC TAGS in the module, requirement for distinct tags will always be satisfied if automatic tagging has been applied

ERROR 2056 OBJECT IDENTIFIER value should have at least two components

`x OBJECT IDENTIFIER ::= { iso }` is wrong object identifier value because encode/decode functions require at least two components for object identifier value

WARNING 2057 Construct #1 has no mapping in SDL

This warning is reported if no mapping to sdl exist but it does not prevent further code generation.

ERROR 2058 Construct #1 has no mapping in SDL

This error indicates that no mapping to sdl exist and is fatal for further code generation.

WARNING 2059 Value given for #1 component

This warning indicates that a value has been given to an optional or default component of an ASN.1 SEQUENCE or SET type. Values for optional and default components cannot be translated to SDL.

WARNING 2060 Constraint could have been extended when mapped to sdl

This warning indicates that constraint transformation has been applied when mapping complex ASN.1 constraints to sdl but the sdl type can allow more values than the ASN.1 type. This can occur when there is no exact mapping of ASN.1 constraints.

ERROR 2061 INTERNAL ERROR in #1

This message indicates an error in the implementation in the utilities. Please send a report to IBM Rational Customer Support, especially if the error can be reproduced as the only error message of an analysis. Contact information for IBM Rational Customer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*

ERROR 2062 Code generation : #1

Error in the generation of SDL, TTCN or encode/decode output.

WARNING 2063 Too big exponent

Exponent in a real value is too big to translate to SDL. This warning message is shown if the exponent is bigger than 1000 or less than -1000.

WARNING 2064 Duplicate synonym name, this synonym will not be mapped to SDL

This message indicates that there are synonym name clashes between named numbers and named bits from INTEGER and BIT STRING types and ASN.1 values if they all will be mapped to SDL (see "Integer" on page 717 and "Bit String" on page 714), and in order to avoid errors only one synonym will be mapped, others are ignored.

ERROR 2065 Number #1 is already assigned to previously defined enumeration item

This error message is reported when NamedNumber alternative is used in an enumerated type definition in an addition enumeration after extension marker and the number #1 has already been assigned to identifier from root enumeration, for example A ::= ENUMERATED {a,b, ... , c(0)} First corresponding numbers are assigned to identifiers in root enumeration, and then in additional enumeration. The above case is invalid, since both 'a' and 'c' are equal to 0.

ERROR 2066 Value for the field '#1' needed #2

This message indicates that value for the field '#1' is missing, but it should be present in #2

ERROR 2067 #1 omitted in #2

Indicates that #1 is omitted, but it should be present in #2

ERROR 2068 #1 should reference #2

This message indicates that a field name references an object class field that is not allowed to be referenced.

ERROR 2069 Wrong object specification

This message indicates that an object specification is incorrect

ERROR 2070 #1 of class #2 needed

This message indicates that an object or object set does not match the governing object class specified in object or object set's definitions

ERROR 2071 Wrong defined syntax

This message indicates that an error in defined syntax for the object definition

ERROR 2072 #1 can not be used in object set specification

This message indicates that an illegal construct is used in the object set specification

ERROR 2073 #1 in the field '#2' is not specified in the #3

This message is reported when information, for example, type or value, is extracted from object field that has not been initialized in the object. This can occur when the field is optional or default in the object class.

WARNING 2074 #1 is not supported in the encoders / decoders

This warning message is reported when ASN.1 notation is used that the encoder / decoder library cannot support

ERROR 2075 #1 can be used only for #2

This message is reported when #1 is used in a component relation constraint but is not allowed to be used in that context.

ERROR 2076 #1

This is used for several different messages concerning component relation constraint, each message is listed and explained below:

Referenced component should refer to the same object class as the referencing one

This message indicates that the referenced and referencing components in a component relation constraint do not stem from the same object class.

Only fixed type value fields are allowed to be specified in a referenced component

This message indicates that a referenced component in a component relation constraint is not a fixed type value field, for example

```
SET {
      a MY-CLASS.@id ({My-set}),
      b MY-CLASS.@TypeField ({My-set})
    }
```

For the field a, @id should reference fixed type value field in class 'MY-CLASS'

Only values of INTEGER types can be used as component relation identifiers";

This message is reported when a referenced fixed type value field is not an INTEGER, only INTEGERS are supported.

```
SET {
      a MY-CLASS.@id ({My-set}),
      b MY-CLASS.@TypeField ({My-set})
    }
```

In the example above @id should be derived from an INTEGER.

Wrong referenced component

This message indicates that a wrong type of component is referenced in a component relation constraint, for example not using ObjectClass-FieldType notation.

Referenced components should be constrained by the same object set as the referencing one

This message indicates that the referenced and referencing component are not constrained with the same object sets.

ERROR 2077 ASN.1 identifier #1 is a keyword, it will be replaced by #2

This message is reported when an ASN.1 identifier is a keyword in one of the target languages and will be changed according to the keywords

configuration file during mapping for avoiding syntax errors in the target languages.

ERROR 2078 Module #1 has got name clashes within joined modules group '#2'

This message is reported when an ASN.1 definition name causes name clashes within joined SDL package and will be prefixed by the original ASN.1 module name during mapping to avoid errors in SDL (see "Joining modules" on page 708).

Restrictions

The ASN.1 Utilities handle all constructs of ASN.1 as defined in ITU-T recommendations X.680, X.681, X.682, X.683, X.690, X.691. There is no support for features defined in the old ASN.1 version X.208 that have been superseded in X.680.

For a list of restrictions see "ASN.1 Utilities" on page 13 in chapter 1, *Compatibility Notes, in the Release Guide*.

Appendix A: List of recognized keywords

By default target language keywords are recognized among ASN.1 identifiers and a postfix `'_<language>_KEYWORD'` is added at the end of the identifier when SDL (`<language> = SDL`), TTCN (`<language> = TTCN`) or C (`<language> = CPP`) is generated. This appendix describes lists of supported keywords for all supported target the target languages.

SDL keywords

```
active, adding, all, alternative, and, any, as, atleast,
axioms, block, break, call, channel, choice, comment,
connect, connection, constant, constants, continue, cre-
ate, dcl, decision, default, else, endalternative, end-
block, endchannel, endconnection, enddecision, endgener-
ator, endmacro, endnewtype, endoperator, endpackage,
endprocedure, endprocess, endrefinement, endselect, end-
service, endstate, endsubstructure, endsyntype, endsys-
tem, env, error, export, exported, external, fi, final-
ized, for, fpar, from, gate, generator, if, import, im-
ported, in, inherits, input, interface, join, literal,
literals, macro, macrodefinition, macroid, map, mod,
nameclass, newtype, nextstate, nodelay, noequality,
```

none, not, now, offspring, operator, operators, optional, or, ordering, out, output, package, parent, priority, procedure, process, provided, redefined, referenced, refinement, rem, remote, reset, return, returns, revealed, reverse, save, select, self, sender, service, set, signal, signallist, signalroute, signalset, size, spelling, start, state, stop, struct, substructure, synonym, syntype, system, task, then, this, timer, to, type, use, via, view, viewed, virtual, with, xor

TTCN keywords

ACTIVATE, AND, BITSTRING, BIT_TO_INT, BY, CANCEL, CASE, COMPLEMENT, CP, CREATE, DO, DONE, ELSE, ENC, ENDCASE, ENDIF, ENDVAR, ENDWHILE, F, FAIL, fail, GOTO, HEXSTRING, HEX_TO_INT, I, IF, IF_PRESENT, INCONC, inconc, INFINITY, INT_TO_BIT, INT_TO_HEX, IS_CHOSEN, IUT, LT, min, MOD, ms, MTC, NOT, ns, OMIT, OR, OTHERWISE, P, LENGTH_OF, none, NUMBER_OF_ELEMENTS, OCTETSTRING, OBJECTIDENTIFIER, PASS, pass, PDU, PERMUTATION, ps, PTC, R, READTIMER, REPEAT, REPLACE, RETURN, RETURNVALUE, R_Type, s, START, STATIC, SUPERSET, SUBSET, THEN, TIMEOUT, TIMER, TO, UNTIL, us, UT, VAR, WHILE

C++ keywords

bool, catch, class, const_cast, delete, dynamic_cast, explicit, false, friend, inline, __multiple_inheritance, mutable, namespace, new, operator, private, protected, public, reinterpret_cast, __single_inheritance, static_cast, template, this, throw, true, try, typeid, typename, using, virtual, __virtual_inheritance, xalloc

auto, asm, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while

Chapter

14 *The CPP2SDL Tool*

The CPP2SDL tool is a C/C++-to-SDL translator that makes it possible to access C or C++ declarations in SDL. The tool takes a set of C/C++ header files as input and generates SDL declarations for a configurable set of the C/C++ declarations in these files.

CPP2SDL is the new generation of the H2SDL utility. Compared to its predecessor, CPP2SDL offers a comprehensive C++ support as well as superior translation configurability. CPP2SDL is fully integrated in SDL Suite, but can also be executed as a stand-alone utility from the command shell.

This chapter is the reference manual for CPP2SDL. The reader is assumed to be familiar with C/C++ and SDL.

Introduction

The overall purpose of the CPP2SDL tool is to provide a convenient means of making external C or C++ declarations available in an SDL context. This is accomplished by translating the C/C++ declarations into representing SDL declarations. These resulting declarations can be injected at an arbitrary level in the SDL scope hierarchy, and may then be used just as if they actually were declared at that scope level. When target code is generated for the SDL system, the Code Generator produces C or C++ code for usages of generated SDL declarations that matches the original C/C++ declarations. The picture below depicts the data flow when using CPP2SDL, and the context of the tool.

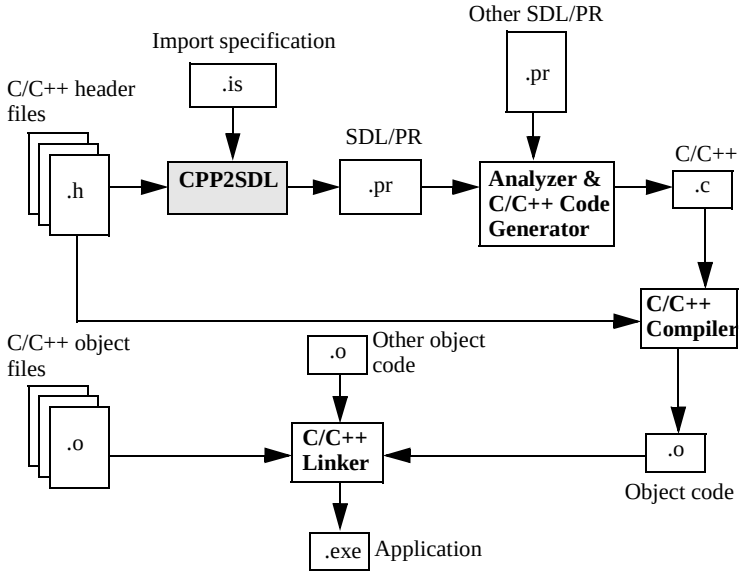


Figure 166 CPP2SDL Data Flow and Context

As can be seen in the figure, the input to CPP2SDL is a set of C/C++ header files and, optionally, an import specification. From this input CPP2SDL generates an SDL/PR file containing SDL representations for the declarations in the header files, or for a subset of these declarations according to what is specified in the import specification. The generated SDL/PR is analyzed together with other SDL/PR, e.g. the

SDL/PR for the SDL system. The Code Generator then generates target C/C++ code which is compiled by a C/C++ compiler. Note that the original C/C++ headers are used in this compilation. The resulting object code is linked together with the object files belonging to the C/C++ headers. Other object files are also included, e.g. the precompiled SDL kernel that is to be used. The result is an executable application.

CPP2SDL translates from C/C++ to SDL according to certain translation rules. These translation rules have been designed to be as simple and intuitive as possible. A user that is familiar with C/C++ should find it straight-forward to use a C/C++ declaration from SDL. The translation rules are described in full detail in "C/C++ to SDL Translation Rules" on page 785. Although CPP2SDL supports translation of a major part of the C and C++ languages, not everything is supported. The limitations of CPP2SDL are listed in "CPP2SDL" on page 27 in chapter 1, *Compatibility Notes*.

Executing CPP2SDL

Normally, CPP2SDL is automatically invoked by the SDL Analyzer as part of the make process. Input header files and tool options are then specified in the Organizer. However, CPP2SDL may also be executed as a stand-alone tool from a command shell, and in that case input headers and tool options are given as command line options.

This section begins with a description of the integration with the Organizer and the Analyzer. Then how to execute CPP2SDL from the command line is described. Finally, follows a section on how to run the tool through the PostMaster.

Execution from the Organizer

The most common way to execute CPP2SDL should be from the Organizer. In fact CPP2SDL will be started automatically by the Analyzer once for each import specification symbol it finds in the Organizer view (see "Import Specifications" on page 776 to learn about import specifications). The Analyzer executes CPP2SDL by means of the PostMaster as described in "Execution from the PostMaster" on page 775. All messages that are output during the execution will be printed in the Organizer Log Window.

Example 80: Executing CPP2SDL from the Organizer ----------

Consider a simple SDL system with one block and one process that needs to access some C++ declarations. At system level certain declarations of the C++ header file `general.h` is used, and at process level declarations of the files `f1.h` and `f2.h` are needed. Figure 167 below shows how the Organizer view of this SDL system could look like.

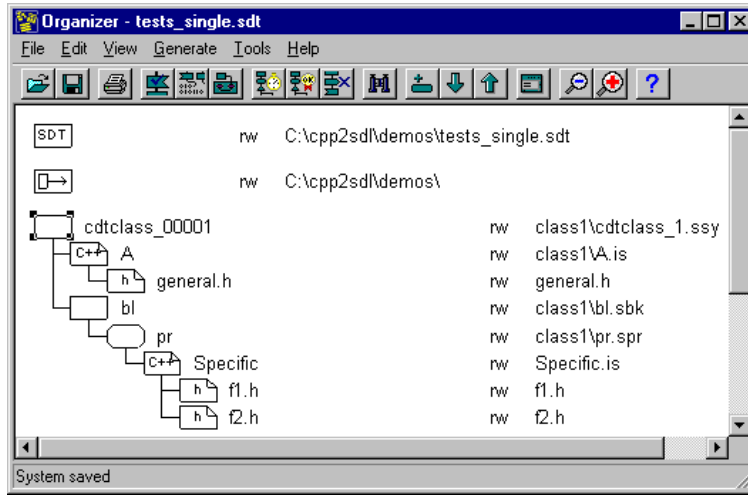


Figure 167 Organizer view with headers to be translated by CPP2SDL

When this system is analyzed, the Analyzer will execute CPP2SDL once for the file `general.h`, and once for the files `f1.h` and `f2.h`. The result of the first translation is a set of SDL declarations that are injected at system level, and thus will be accessible in all scopes. The result of the second translation is a set of SDL declarations that are injected at process level and thus are not accessible in the system or in the block scope.

Adding Import Specifications to the Organizer view

The first step in accessing C/C++ declarations from SDL is to insert a PR symbol at the place in the SDL specification where the C/C++ declarations are to be used. The PR symbol represents the inclusion of the SDL PR that is the translation of the C/C++ declarations.

To specify that this should be an import specification, double-click the PR symbol either in the Organizer or in the SDL Editor to open the Edit Document dialog. In the dialog it is possible to select either C Import Specification or C++ Import Specification.

An import specification can be edited manually by means of the Text Editor (see "Import Specifications" on page 776 to learn about import

specifications). However, an import specification can also be edited in the CPP2SDL Options dialog described below.

After adding an import specification it is necessary to specify which C/C++ header files are to be translated. This is done by selecting the import specification in the Organizer and then use the Add Existing and Add New commands to select or create C/C++ header files respectively.

Setting CPP2SDL Options in the Organizer

Required options to CPP2SDL may be specified in the Organizer for each import specification by using the CPP2SDL Options dialog. This dialog may be opened from the menu that appears when the right mouse button is pressed on an import specification symbol. Figure 168 shows this dialog.

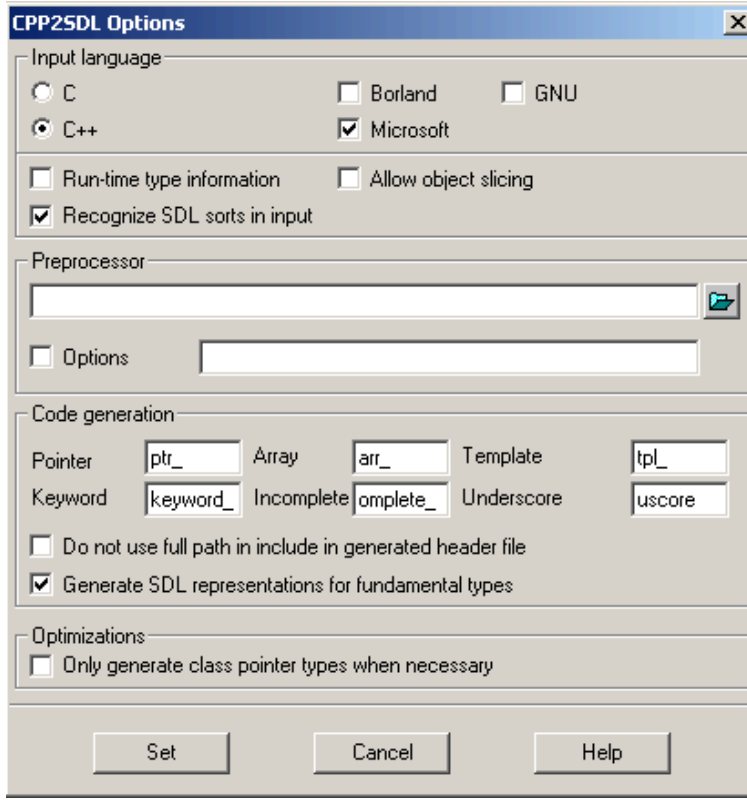


Figure 168 The CPP2SDL Options Dialog

The fields and buttons of the CPP2SDL Options dialog correspond directly to the command line options described in "Command Line Options" on page 769:

- *Language*

  These radio buttons select the input language. If C is selected, CPP2SDL will be executed in C mode, i.e. as with the `-c` command line option.

- *Dialect*

These check boxes determine what dialects to support in the input, and correspond to the `-dialects` command line option. If no check-boxes are marked, the ANSI C/C++ dialect is supported.

- *Run-Time Type Information*

  This check box should be set if Run-Time Type Information (RTTI) is available in C++ and should be supported in the SDL translation. It corresponds to the `-rtti` command line option.

- *Allow object slicing*

  This check box should be set if object slicing should be supported in the SDL translation. It corresponds to the `-slicing` command line option.

- *Recognize SDL sorts in input*

  This check box should be set if SDL sorts should be recognized in the input. It corresponds to the `-sdlsorts` command line option.

- *Preprocessor*

  This field is used to specify the preprocessor to use for preprocessing the input. It corresponds to the `-preprocessor` command line option. This field also has a browse button that makes it possible to select the preprocessor from a file selection dialog.

- *Preprocessor options*

  This field should contain the options to the preprocessor. It corresponds to the `-cppoptions` command line option.

- *Pointer, Array, Template, Keyword, Incomplete, Underscore*

  These fields specify the prefixes and suffixes that are used when C/C++ names must be modified in the SDL translation. They correspond to the `-prefix` and `-suffix` command line options.

- *Do not use full path in include in generated header file*

  Checking the box will prevent usage of path in the includes and make it easier to move files to other directory locations. It corresponds to the `-noabsolutepath` command line option.

- *Generate SDL representations for fundamental types*

This check box should be set if SDL representations for fundamental C/C++ types should be included in the translation. It corresponds to the `-generatecpptypes` command line option.

- *Only generate class pointer types when necessary*

  If this check box is set, CPP2SDL will optimize the generation of class pointer types. It corresponds to the `-optclasspointers` command line option.

Execution from Command Line

CPP2SDL is invoked from the command prompt by the command:

```
cpp2sdl [options] <C/C++ header files>
```

Unless the `-post` option is set, all messages that are output by the tool, e.g. errors and warnings, will be printed on the standard error stream (`stderr`).

CPP2SDL will translate the declarations in the specified C/C++ header files, or a subset of these declarations if a suitable import specification is used (see "Import Specifications" on page 776). The resulting SDL declarations will be saved in a file called *`name`*`.pr`, where *`name`* is the name of the import specification used. If no import specification is used, *`name`* will be the name of the first input header file.The output file will be placed in the same directory from where CPP2SDL is executed.

There is a possibility when running in batch mode to be able to change the CPP2SDL options used in an SDL system. See "Batch Facilities" on page 208 in chapter 2, *The Organizer*.

Normally CPP2SDL options are saved in an import ("`.is`") file. These options will in batch be superseded by CPP2SDL options given in a batch "`.sdt`" option file.

Command Line Options

The command line options recognized by CPP2SDL are listed and explained below. Note that an option may be abbreviated as indicated by the underlined part of the option name.

- `-append`

Append the generated SDL declarations to the file that is specified with the `-output` option. If that file does not exist, this option will be ignored and CPP2SDL will create a new file for the output as usual.

- `-c`

  Execute in C mode. CPP2SDL will assume that no C++ specific constructs are encountered in the input headers. If this assumption does not hold, the result of the translation is undefined. See "Special Translation Rules for C Compilers" on page 846 for a detailed description of translation rule modifications that are caused by using this option.

- `-cppoptions <optionsstring>`

  Send the specified option string to the preprocessor. If the string contains white spaces, it must be quoted.

- `-dialects <dialect> <dialect> ... <dialect>`

  Accept the specified C/C++ dialects in the input headers. Supported dialects are

  – ANSI (ANSI C/C++)

  – GCC (Gnu C/C++)

  – MSVC (Microsoft Visual C/C++)

  – ALL (all supported C/C++ dialects)

  If this option is not used, CPP2SDL will assume that the input headers conform to the ANSI C/C++ dialect.

- `-errorlimit <number>`

  Set the maximum number of errors to report before terminating the translation. The default it to terminate when 5 errors have been found.

- `-extsyn`

  Will not generate for constants with numeric expressions, external synonyms with its value (if the expression can be calculated during translation). Default, i.e. without this option, the value is translated.

- `-generatecpptypes`

Include SDL representations for fundamental C/C++ types in the translation. See "SDL Library for Fundamental C/C++ Types" on page 848 for more information about what actually is generated when this option is used.

- `-help`

  Print a help message about CPP2SDL. No translation will be performed.

- `-importspecification <file>`

  Use the specified file as import specification for the translation. Import specifications are described in "Import Specifications" on page 776.

- `-noabsolutepath`

  Prevent usage of path in the includes to make it easier to move files to other directory locations.

- `-nocheckinput`

  Do not check that all input headers are existing and readable before trying to translate them. The use of this option could make it easier to use CPP2SDL from scripts.

- `-nodepend`

  Do not translate depending declarations when using an import specification. Only the identifiers that are explicitly present in the import specification will be translated. If this option is set, CPP2SDL cannot guarantee that the resulting set of SDL declarations is complete and consistent. See "Import Specifications" on page 776 for more information.

- `-noinclude`

  Ignore definitions #included into imported header file by the preprocessor. Only definitions that are explicitly present in the original header file will be translated, and may be some others if the -nodepend option is not set.

  This option allows having several separate import specifications for several imported header files, even if these files include each other. Although, conflicts in SDL are still possible if the same depending definitions are imported for different import specifications. So, to achieve the best result, the -noinclude option should often be used

together with the -nodepend option. Both options are not present in "CPP2SDL Options..." dialog window and should be specified directly in the *.is file.

To avoid build errors when using several import specifications for several header files that include each other, do not forget to use protection macros #ifndef-#define-#endif in the header files.

- `-novariables`

  Do not generate external variables. This option is needed since the rules for where SDL allows declarations of external variables are more restrictive than for other declarations. For example, SDL does not allow external variables declared at system or block level. If this option is used, CPP2SDL will output a warning if it finds a construct that otherwise would be translated to an external variable.

- `-optclasspointers`

  Optimize the generation of class pointer types so that they are only generated when they appear in the input headers. If this option is not used, CPP2SDL will automatically generate a pointer type to all translated classes. Read more about this in "Classes, Structs and Unions" on page 803.

- `-output <file>`

  Write the resulting SDL declarations to the specified file. If the `-append` option is set, the result will be appended to the file. Otherwise a new file will be created, overwriting an existing file with the same name, if any.

  Note that all files that CPP2SDL generates will be placed in the same directory as the generated SDL/PR file.

- `-post`

  Start CPP2SDL as a PostMaster client waiting for requests from the PostMaster. The PostMaster messages that are handled by CPP2SDL are described in "Execution from the PostMaster" on page 775.

- `-prefix "ptr=<string> arr=<string> keyword=<string> incomplete=<string> tpl=<string>"`

  Use the specified name prefixes when generating SDL. CPP2SDL uses name prefixes when the original C/C++ names for some reason cannot be used in SDL. This option makes it possible to fully con-

figure how such modified names are generated. This is often useful in order to avoid name clashes in SDL.

- `-preprocessor <executable>`

  Use the specified executable for preprocessing the input headers. The executable should be a preprocessor or C/C++ compiler that is supported by CPP2SDL:

  – 'cl' (Microsoft Visual C/C++ Compiler), **in Windows**.

  – 'cpp' (C/C++ Preprocessor), **on Unix**.

  – 'cc' and 'CC' (Sun Workshop C and C++ Compilers), **on Unix**.

  – 'gcc' and 'g++' (GNU C and C++ Compilers), **on Unix**.

  If this option is not used, CPP2SDL will attempt to use 'cl' **in Windows**, and 'cpp' **on Unix**.

  Note that CPP2SDL uses name matching of the specified filename, with the file name extension stripped, to determine what preprocessor or compiler to use for preprocessing. If the specified name does not match the name of any supported preprocessor or compiler on the current platform, CPP2SDL will attempt to call the executable like this:

  ```
  <executable> <options> <input file> <output file>
  ```

  `<options>` are the option string specified with the `-cppoptions` options.
  If this call fails, CPP2SDL does not know how to preprocess the input headers and terminates.

Hint:

If you want to preprocess the input headers using a preprocessor that is not supported by CPP2SDL, you can write a simple shell script that wraps the call to the desired preprocessor. The script should conform to the call style that CPP2SDL uses for unknown preprocessors. Then execute CPP2SDL, using the `-preprocessor` option to specify the script as the preprocessor to use.

- `-ref`

Include source references in the generated SDL. The format of these source references is described in "Source and Error References" on page 783.

- `-rtti`

  Assume Run-Time Type Information, and support dynamic casting. See "Run-Time Type Information and Dynamic Cast" on page 827 for more information what this means.

- `-sdlsorts`

  Recognize SDL sorts in input. CPP2SDL will translate C/C++ types that are prefixed with 'SDL_' to the corresponding SDL sort. Refer to "SDL Sorts in C/C++" on page 844 for an example on how this feature can be used.

- `-slicing`

  Generate SDL cast operators to support slicing of C++ objects. See "Type Compatibility between Inherited Classes" on page 823 for more information.

- `-sortmembers`

  Sort struct members in SDL newtypes alphabetically.

- `-suffix "uscore=<string>"`

  Use the specified name suffixes in the generated SDL. CPP2SDL uses name suffixes when the original C/C++ name for some reason cannot be used in SDL. This option makes it possible to fully configure how such modified names are generated.

- `-targetdir <directory>`

  Set the target directory for generated files. CPP2SDL produces one single header file which includes all the header files that are to be translated. If this option is used, this generated header file is placed in the specified target directory. Otherwise the file will be placed in the same directory as the generated SDL/PR file.

- `-version`

  Show version information.

Example 81: Executing CPP2SDL from the command line

```
% cpp2sdl -preprocessor /usr/ccs/lib/cpp -output
result.pr -prefix "ptr=p arr=a" -rtti -ref input.h
```

This command will translate the input header `input.h` to SDL and write the resulting SDL declarations to the file `result.pr`. The specified preprocessor 'cpp' will be used to preprocess the input. If the input contains pointer or array types, the corresponding SDL names will be prefixed with 'p' and 'a' respectively. Source references to the declarations in `input.h` will be generated by CPP2SDL, and Run-Time Type Information is assumed so that dynamic cast operators are generated.

Execution from the PostMaster

As mentioned above, CPP2SDL may be started as a PostMaster client by using the `-post` option at the command line. As a PostMaster client, CPP2SDL will handle two different PostMaster events.

- SESTOP
- SECPP2SDLCOMMAND <optionstring>

The reception of a SESTOP event has the expected behavior; CPP2SDL ceases to be a PostMaster client and terminates.

The SECPP2SDLCOMMAND event has an option string as argument. The event will cause CPP2SDL to execute according to the options specified in that string. The format of the option string is the same as when CPP2SDL is executed from the command line (see "Execution from Command Line" on page 769). All messages that are output by the tool will be broadcast to the PostMaster.

After the execution of a SECPP2SDLCOMMAND event a reply is sent:

SECPP2SDLCOMMANDREPLY <#errors> <#warnings> <status>

The <#errors> and <#warnings> arguments tell the number of errors and warnings that occurred during the translation, and <status> is a text string with the same information in a more readable form.

Example 82: Executing CPP2SDL from the PostMaster

A single PostMaster may be started with this command:

```
% sdt -noclients
```

Then CPP2SDL is started as a PostMaster client:

```
% cpp2sdl -post &
```

CPP2SDL is now waiting for requests to come from the PostMaster. By using for example the SERVERPC application, events can be sent to it.

```
% serverpc 58000 58101 "-rtti -ref input.h"
```

58000 is the tool id of CPP2SDL, and 58101 is the event id for the SECPP2SDLCOMMAND event. As a result the following reply event could for example be received:

SECPP2SDLCOMMANDREPLY 0 2 “0 errors and 2 warnings”

Finally, CPP2SDL is terminated using the SESTOP event (id 58303):

```
% serverpc 58000 58303
```

Import Specifications

A simple but powerful way of configuring the translation of a set of C/C++ declarations is to use an import specification. As the name suggests, an import specification specifies how to import external code into SDL. An import specification for CPP2SDL is a textfile written in a simple C/C++-style syntax. The file consists of two sections that both are optional:

- `CPP2SDLOPTIONS`

  This section may contain options to the CPP2SDL tool. The syntax is the same as when CPP2SDL is executed as a stand-alone tool from the command line. See “Command Line Options” on page 769.

- `TRANSLATE [ALL]`

  This section may contain a list of C/C++ identifiers. CPP2SDL will attempt to make these identifiers available in SDL by translating the corresponding declarations.

  Optional keyword ALL tells that, in spite of provided list of identifiers, all objects from the source files should be imported.

Options and identifiers in an import specification are delimited by newlines.

The example below shows a simple import specification where the identifiers `func`, `C` and `myint` are made available in SDL.

Example 83: A simple import specification ––––––––––––––––––––

```
CPP2SDLOPTIONS {
   -preprocessor /usr/ccs/lib/cpp
}

TRANSLATE {
   func
   C
   myint
   // C++ style comment (if supported by preproc.)
   /* C style comment */
}
```

––

The import specification file will be preprocessed by CPP2SDL with the same preprocessor and preprocessor options that are used when preprocessing the input C/C++ headers. This makes it possible to use, for instance, C/C++ comments and macros in an import specification.

> **Note:**
>
> Some preprocessors will refuse to preprocess files that have an unknown file name extension (for example `.is`). In that case the import specification file must be given a file name extension that is known to the preprocessor (for example `.h`).

If an identifier in an import specification refers to a declaration that depends on other declarations, CPP2SDL will, by default, translate all these depending declarations as well. This principle is applied recursively to all declarations that depend on depending declarations, thereby making sure that the resulting SDL declarations are complete and consistent. If the `-nodepend` option is set, depending declarations will not be translated automatically. Then the tool cannot guarantee that the resulting set of SDL declarations is complete and consistent.

Example 84: Translation of depending declarations––––––––––––

File `data.h`:

```
typedef int myint;

class C {
public:
  myint* mvar;
```

```
};
```

File `import.is`:

```
TRANSLATE {
   C
}
```

Execution of CPP2SDL from the command line,

```
% cpp2sdl -importspecification import.is data.h
```

will produce resulting SDL declarations in the file `import.pr`:

```
SYNTYPE myint = int
ENDSYNTYPE myint;EXTERNAL 'C++';
NEWTYPE ptr_myint Ref( myint);
  OPERATORS
    ptr_myint : -> ptr_myint;
    ptr_myint : ptr_myint -> ptr_myint;/
ENDNEWTYPE ptr_myint;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  STRUCT
    mvar ptr_myint;
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

Here, the import specification only specifies that the class `C` shall be translated, but since the declaration of `C` depends on `myint*`, which in turn depends on `myint`, these declarations will be translated as well.

Note:

CPP2SDL will only translate those identifiers in an import specification that refer to declarations in namespaces (including the global namespace). If an identifier refers to another kind of declaration, for example a class member, it will be ignored and CPP2SDL will issue a warning.

Advanced Import Specifications

Besides from specifying which identifiers that should be translated to SDL, there are some more advanced constructs that may be used in an import specification.

Type Declarators

It is possible to append type declarators to identifiers that represent types. The same type declarators as in C/C++ are allowed, i.e. pointer (*), array ([]), and reference (&). Prefix and postfix declarators are separated by a dot (.).

Example 85: Type declarators in import specification ––––––––––––

```
TRANSLATE {
   char*         // A pointer to char.
   MyClass.[8]   // An array of 8 MyClass.
   mytype&       // A reference to mytype.
   C*.[10]       // An array of 10 pointers to C.
}
```

––

Prototypes for Ellipsis Functions

The translation rule for a function with unspecified arguments (a.k.a an ellipsis function) requires that information is provided about which versions of the function that should be made available in SDL (see "Unspecified Arguments" on page 796). This information may be given in an import specification by specifying prototypes for the function.

Example 86: Prototypes for ellipsis functions in import specification

Input declaration:

```
int printf(const char*, ...);
```

Import specification:

```
TRANSLATE {
   printf
   printf(int)
   printf(double, char)
}
```

Resulting SDL declarations:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
```

```
  OPERATORS
    printf : ptr_char, double, char -> int;
    printf : ptr_char, int -> int;
    printf : ptr_char -> int;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';

NEWTYPE ptr_char Ref( char);
  OPERATORS
    ptr_char : -> ptr_char;
    ptr_char : ptr_char -> ptr_char;/
ENDNEWTYPE ptr_char;EXTERNAL 'C++';
```

Template Instantiations

Similar to ellipsis functions, the translation of templates requires additional information about how the templates should be instantiated (see "Templates" on page 838). This information may be specified in an import specification, using the same syntax as when templates are instantiated in C++.

Example 87: Template instantiations in import specification

Input declarations:

```
template <class C, int i> class S {
public:
  C arr[i];
};

template <class D> D func(const D& p1);
```

Import specification:

```
TRANSLATE {
   S<double, 5>
   func<unsigned int>
}
```

Resulting SDL declarations:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    tpl_func_unsigned_int /*#REFNAME 'func<unsigned
int >'*/ : unsigned_int -> unsigned_int;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';

NEWTYPE arr_5_double CArray( 5, double);
ENDNEWTYPE arr_5_double;EXTERNAL 'C++';

NEWTYPE ptr_tpl_S_double_5 Ref( tpl_S_double_5);
  OPERATORS
```

```
    ptr_tpl_S_double_5 : -> ptr_tpl_S_double_5;
    ptr_tpl_S_double_5 : ptr_tpl_S_double_5 ->
    ptr_tpl_S_double_5;
ENDNEWTYPE ptr_tpl_S_double_5;EXTERNAL 'C++';

NEWTYPE tpl_S_double_5 /*#REFNAME 'S<double, 5 >'*/
  STRUCT
    arr arr_5_double;
  OPERATORS
    tpl_S_double_5 /*#REFNAME 'S'*/ : ->
    tpl_S_double_5;
   tpl_S_double_5 /*#REFNAME 'S'*/ : tpl_S_double_5
-> tpl_S_double_5;
ENDNEWTYPE tpl_S_double_5;EXTERNAL 'C++';
```

Note that since class template instantiations define types, it is possible to use type declarators for them.

Providing additional import information in the TRANSLATE section

For the cases when all objects from the input header files should be imported, and at the same time additional information for the import should be provided (for example, information on ellipsis function prototypes), TRANSLATE ALL section can be used.

The "TRANSLATE ALL { }" section contains a list of objects, but it does not limit the number of translated objects to the listed ones. All objects from the source file are translated, the translate section provides additional information for translation of advanced objects, such as ellipsis functions, templates, etc.

Example 88: Use of TRANSLATE

f1.h contains types X, Y and f2.h contains types Z, printf

When translating both header files with the following import specification:

```
TRANSLATE {
  X
  printf(int)
}
```

the imported PR Files will contain type X and one prototype of printf(...)

For the import specification:

```
TRANSLATE ALL {
  X
  printf(int)
}
```

the imported PR file will contain all types (X, Y, Z) and one prototype for the ellipsis function

---

Source and Error References

A source reference is a reference from a generated SDL declaration to the corresponding original C/C++ declaration. Source references are placed in the generated SDL/PR file.

An error reference is also a reference to a declaration in the input header file, but is used to point out an error (or a warning) in that file. Error references are therefore printed as messages to the standard error stream or to the Organizer Log Window.

CPP2SDL uses the #SDTREF format both for source and error references. See chapter 18, *SDT References* for more about #SDTREF.

Source References

When CPP2SDL is executed from the Organizer, or from the command line with the `-ref` option set, the generated SDL/PR file will contain references to the input source files. Such a reference occurs just before a generated SDL declaration, and is on the form

```
/*#SDTREF(TEXT,filename,line)*/
```

where

- `filename` is the name of the input file where the corresponding C/C++ declaration can be found.
- `line` is the line number in that input file where the C/C++ declaration starts.

A source reference is shown in Example 89 below.

Example 89: Source references ----------

```
/*#SDTREF(TEXT,input.h,226)*/
NEWTYPE S STRUCT
  a int;
OPERATORS
  get_a: S -> int;
ENDNEWTYPE S;EXTERNAL 'C++';
```

Error References

Error references have a similar format as source references but with a column position added after the line number:

```
/*#SDTREF(TEXT,filename,line,column)*/
```

CPP2SDL prints error references when errors or warnings are found during the translation. They are output to the standard error stream (`stderr`) or to the Organizer Log Window depending upon whether CPP2SDL is executed from the command line or from the PostMaster.

Problems with error references may arise because of the preprocessor. Among other things, the preprocessor expands macros, and a typical problem is illustrated in Example 90 below.

Example 90: Error References ––––––––––––––––––––––––––––––

File `def.h`:

```
#define init InitializingFunction

void init(undefinedType *, int);
```

If CPP2SDL translates this file, the following error message will be printed:

```
#SDTREF(TEXT,def.h,3,27)
ERROR 3200 Syntax error.
```

Here the syntax error occurs at position (3,11) in the source file, but because of the macro expansion of `init` to `InitializingFunction`, CPP2SDL will report the error at position (3,27) instead. Thus the column position is several characters off the target in the original file. When using the Organizer Log's *Show Error* function (see "Show Error" on page 187 in chapter 2, *The Organizer*) to view the source of this error message, the cursor will be placed at `int` instead of at `undefinedType`. CPP2SDL calculates both source and error references from the preprocessed source code, and this may lead to reference problems when macros are involved.

––

C/C++ to SDL Translation Rules

The general idea behind the CPP2SDL tool is to take a set of C/C++ header files, preprocess them, and translate some or all of the declarations in these headers into SDL/PR representations. This section describes the rules for this translation process.

Each C/C++ construct is described in a subsection of its own. First, a general rule for the translation of the construct is presented. Then follows a description of exceptions to this rules, and rationals for these exceptions.

Before proceeding, it should be noted that the translation rules have been designed to support both C and C++ target compilers. To a large extent the translation rules are actually independent of whether a C or C++ target compiler is used. However, there are some differences, so when CPP2SDL executes in "C mode" (i.e. with the `-c` option set) a few translation rules are slightly modified. These modifications are described in "Special Translation Rules for C Compilers" on page 846.

Names

Rule: The name of a C/C++ identifier is the same in SDL.

The naming rules of identifiers in SDL and C/C++ are rather similar but differs in two important aspects:

- SDL is a case-insensitive language, while C/C++ is case-sensitive.
- SDL has some restrictions for how underscores may be used in names. C/C++ has no such restrictions.

To overcome these differences tool specific extensions have been made in the supported SDL dialect. The Analyzer has an option to handle case sensitive SDL (see "Set-Case-Sensitive" on page 2486 in chapter 54, *The SDL Analyzer*), and most of the restrictions with underscores have been removed. However, the rule that a name that ends with an underscore should be concatenated with the following name, makes it necessary to modify such names in the SDL mapping. This is done by appending a string suffix to such names. This string is by default "uscore" but may be configured to an arbitrary string be means of the CPP2SDL option
`-suffix`.

Another case where a name in C/C++ cannot be retained in the SDL translation is when the name is an SDL keyword. Such names are prefixed with a user-configurable string that by default is `keyword_`. The option `-prefix` can be used to configure this string.

Example 91 below gives some examples of the translation rules for names.

Example 91: Translation of names ----------

C++:

```
int ABC, abc;   // Case sensitivity
char u__sc, _w, x_; // Unrestricted use of
underscores
double signal;  // SDL keyword
```

SDL:

```
DCL ABC int; EXTERNAL 'C++';
DCL abc int; EXTERNAL 'C++';
DCL u__sc char; EXTERNAL 'C++';
DCL _w char; EXTERNAL 'C++';
DCL x_uscore /*#REFNAME 'x_'*/ char; EXTERNAL 'C++';
DCL keyword_signal /*#REFNAME 'signal'*/ double;
EXTERNAL 'C++';
```

Note the #REFNAME directive that passes the original C/C++ name to the Code Generator for names that are modified in the SDL translation.

Fundamental Types

Rule: A fundamental C/C++ type is mapped to an SDL sort with the same name.

The SDL sorts that represent fundamental C/C++ types are not generated by CPP2SDL but are defined in special SDL/PR files that are included if the `-generatecpptypes` option is set. The SDL sorts in these files are normally syntypes of predefined SDL sorts. Refer to "SDL Library for Fundamental C/C++ Types" on page 848 for more information.

The table below shows how the fundamental C/C++ types are translated to SDL sorts, and what predefined SDL sort that correspond to these SDL sorts.

C/C++ Fundamental Type	SDL Sort	Predefined SDL Sort
signed int int	int	Integer
unsigned int unsigned	unsigned_int	Integer
signed long int signed long long int long	long_int	Integer
unsigned long int unsigned long	unsigned_long_int	Integer
signed short int signed short short int short	short_int	Integer
unsigned short int unsigned short	unsigned_short_int	Integer
signed long long int signed long long long long int long long	long_long_int	Integer
unsigned long long int unsigned long long	unsigned_long_long_int	Integer
char	char	Character
signed char	signed_char	Character
unsigned char	unsigned_char	Octet
wchar_t	wchar_t	N/A
float	float	Real
double long double	double	Real
bool	bool	Boolean
void	N/A	N/A

Note that the `wchar_t` type has no counterpart in SDL, and thus is represented by a newtype rather than a syntype of a predefined sort.

Also note that the special `void` type is not represented explicitly in SDL. Instead this type is translated by omitting input and result arguments to operators as described in "Functions" on page 793.

Type Declarators

There are two different type declarators in C; pointer (*), and array ([]). C++ has one additional type declarator; reference (&).

Pointers

Rule: A type with a pointer declarator is translated by applying the `Ref` generator on the SDL sort that corresponds to that type.

The name of the generated newtype for the pointer is prefixed with a user-configurable string that by default is `ptr_`. The option `-prefix` can be used to configure this string.

Untyped pointers (`void*`) are translated to a special SDL sort called `ptr_void`. See "SDL Library for Fundamental C/C++ Types" on page 848 for more information about the `Ref` generator and the special `ptr_void` sort.

Example 92: Translation of pointers ----------

C++:

```
typedef int* p_int;
extern void* generalp;
```

SDL:

```
NEWTYPE ptr_int Ref( int);
  OPERATORS
    ptr_int : -> ptr_int;
    ptr_int : ptr_int -> ptr_int;
ENDNEWTYPE ptr_int;EXTERNAL 'C++';
SYNTYPE p_int = ptr_int
ENDSYNTYPE p_int;EXTERNAL 'C++';
DCL generalp ptr_void; EXTERNAL 'C++';
```

Arrays

Rule: A type with an array declarator is translated by applying the `CArray` generator on the SDL sort that corresponds to that type.

There is one important exception to this rule. Array declarators that do not specify the size of the array are translated in the same way as pointers (see "Pointers" on page 788).

The name of the generated newtype for an array type with a specified size is prefixed with a user-configurable string that by default is "arr_". The option `-prefix` can be used to configure this string. The name also contains the size of the array, since the size is used in the `CArray` generator instantiation and thus is significant in SDL. This makes SDL array sorts of different sizes type incompatible, but this is normally not a big problem since the elements of the arrays are type compatible.

> **Note:**
>
> SDL array sorts corresponding to C/C++ arrays with different sizes are normally type incompatible.

Example 93: Translation of arrays ----------

C++:

```
extern char c_arr1[20];
extern char c_arr2[];
```

SDL:

```
NEWTYPE arr_20_char CArray( 20, char);
ENDNEWTYPE arr_20_char;EXTERNAL 'C++';
DCL c_arr1 arr_20_char; EXTERNAL 'C++';
NEWTYPE ptr_char Ref( char);
  OPERATORS
    ptr_char : -> ptr_char;
    ptr_char : ptr_char -> ptr_char;
ENDNEWTYPE ptr_char;EXTERNAL 'C++';
DCL c_arr2 ptr_char; EXTERNAL 'C++';
```

References

Rule: A type with a reference declarator is translated as normal, i.e. the reference declarator is not translated to SDL.

A C++ reference can be looked upon as a constant pointer that is automatically de-referenced each time it is used. This makes a reference an alternative name for an object. Since no difference will be made between an object and a reference to an object in the SDL mapping, references will appear to be objects in SDL.

Example 94: Translation of references ----------

C++:

```
extern int i; /* i is initialized elsewhere */
extern int& r; /* r is initialized to i elsewhere
                                              (int& r =
i;), i.e. r and i refers to
                    the same int. */
```

SDL:

```
DCL i int; EXTERNAL 'C++';
DCL r int; EXTERNAL 'C++';/* N.B. C++ reference! */
```

Note that if `r` in this example is assigned a value in SDL, it is in fact the object that `r` refers to (i.e. `i`) that gets a new value. This could be confusing if only the SDL translation of `r` is considered, and to avoid this a comment is attached to the declaration of `r` that tells that it is a reference in C++.

References could also appear as specifiers for formal function arguments. Such arguments will be translated to operator arguments marked with the IN/OUT keyword if they are non-constant (see "Argument Passing and Return Value" on page 794).

Enumerated Types

Rule: An enumerated type is translated to a newtype with literals corresponding to the enum literals.

A special case is when the enumerated type has no literals. Such a type can be treated as an integer in C/C++, and is consequently translated to a syntype of `int`.

Example 95: Translation of enumerated types----------

C/C++:

```
enum {} v;
enum E2 {};
enum E1 {a, b, c=10};
```

SDL:

```
DCL v int; EXTERNAL 'C++';
SYNTYPE E2 = int
ENDSYNTYPE E2;EXTERNAL 'C++';
NEWTYPE E1
```

```
    LITERALS a, b, c;
    OPERATORS
      IntToEnum /*#REFNAME '(E1)'*/ : int -> E1;
      EnumToInt : E1 -> int; /*#OP(PY)*/
      ORDERING;
ENDNEWTYPE E1;EXTERNAL 'C++';
```

By using the “type conversion” operators `EnumToInt` and `IntToEnum` integer arithmetic and comparisons become available in SDL also for enumerations. As can be seen from the #REFNAME directive in the example above, the generated code for calls to the `IntToEnum` operator will be a C style cast from `int` to `enum`. This explicit type conversion should be acceptable by all target compilers. Also note that the #OP(PY) directive means that there will be no generated code for calls to the `EnumToInt` operator, which is desired since that type conversion is implicit in C/C++.

If the enumerated type is incomplete, i.e. if the enum tag is missing, the translation rule is slightly modified according to the translation rules for incomplete types (see “Incomplete Types” on page 830). For enumerations, these rules have the following impact:

· The name of the generated newtype follows the naming rules for incomplete types described in “Incomplete Types” on page 830.

· The newtype will not be external, since it does not correspond to a C/C++ type that may be referred to.

· The `IntToEnum` operator will not be generated for the same reason.

Typedef Declarations

Rule: A typedef declaration is translated to an SDL syntype declaration.

There are two exceptions to this rule:

- A typedef declaration of a tagged type[1], where the typedef name is the same as the name of the tag.
- A typedef declaration where the typedef name has been omitted. This is a legal but not very common case.

1. A tagged type is a class, struct, union or enum type with a tag.

In these cases the typedef declarations do not define new typenames, and thus no syntypes need to be generated.

Another special case is when a synonym for `void` is introduced by means of a typedef declaration. Such a typedef declaration is not translated, but the typedef name will be remembered. References to the typedef name will then be translated in the same way as `void` would have been translated in that context.

Example 96: Translation of typedef declarations ––––––––––––––––

C++:

```
typedef int MyInt;
typedef struct r {
                int a;
} r; // Typedef name is the same as the tag name!
typedef struct s {
  MyInt a;
}; // Omitted typedef name - legal but rare!
typedef void myvoid;
typedef myvoid myvoid2;
myvoid f(myvoid2);
```

SDL:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    f :;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';
SYNTYPE MyInt = int
ENDSYNTYPE MyInt;EXTERNAL 'C++';
NEWTYPE ptr_r Ref( r);
  OPERATORS
    ptr_r : -> ptr_r;
    ptr_r : ptr_r -> ptr_r;
ENDNEWTYPE ptr_r;EXTERNAL 'C++';
NEWTYPE r
  STRUCT
    a int;
  OPERATORS
    r : -> r;
    r : r -> r;
ENDNEWTYPE r;EXTERNAL 'C++';
NEWTYPE ptr_s Ref( s);
  OPERATORS
    ptr_s : -> ptr_s;
    ptr_s : ptr_s -> ptr_s;
ENDNEWTYPE ptr_s;EXTERNAL 'C++';
NEWTYPE s
  STRUCT
    a MyInt;
  OPERATORS
```

```
    s : -> s;
    s : s -> s;
ENDNEWTYPE s;EXTERNAL 'C++';
```

Note:

Typedefs of function types are not supported by CPP2SDL, and will not be translated to SDL.

Functions

Rule: A function prototype is translated to an SDL operator signature.

This rule is valid both for member and non-member functions. Operators that result from functions that are members of a class will be placed in the newtype that is the translation of that class. Operators that result from non-member functions will be placed in a special newtype called `global_namespace_ImpSpec`[1].

Member functions are described in "Members" on page 807, and the rest of this section will focus on non-member functions.

Example 97: Translation of non-member functions

C++:

```
char myfunc1(char);
int myfunc1();
void myfunc2();
void myfunc2(int);
```

SDL:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    myfunc1 : char -> char;
    myfunc1 : -> int;
    myfunc2 :;
    myfunc2 : int;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';
```

Note that functions without input arguments or return value, will be translated to operators without input arguments or return value. Such operators are not allowed according to the SDL96 standard, but are accepted by the SDL Analyzer as a tool specific language extension.

1. This name indicates that the newtype represents the global scope in C/C++. In C++ terminology this scope is often called the global namespace.

Overloaded Functions

Rule: Overloaded functions are translated to overloaded SDL operators.

The semantics of overloaded functions in C++ differs slightly from the semantics of overloaded operators in SDL. For example, C++ allows overloading on constant arguments which is not possible in SDL. A C++ header file may therefore contain overloaded functions that cannot be translated to SDL. Normally this is not a problem since the C++ compiler resolves generated calls to these functions correctly anyway.

Example 98: Translation of overloaded functions

C++:

```
int f0();
int f0(double);
int f1(int&);
int f1(const int&);
```

SDL:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    f0 : -> int;
    f0 : double -> int;
    f1 : int -> int;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';
```

Argument Passing and Return Value

Rule: Function arguments that are passed by reference are translated to IN/OUT operator arguments in SDL.

There is one exception to this rule; arguments that are references to constants do not translate to IN/OUT arguments since C++ allows these arguments to take variables as well as constant values.

Example 99: Translation of function arguments and return value

C++:

```
int f1 (int p1, int &p2, const int &p3, const int
*p4, int *const p5);
int &f2();
const int &f3();
```

SDL:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    f1 : int, IN/OUT int, int, ptr_int, ptr_int ->
int;
    f2 : -> int;
    f3 : -> int;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';
NEWTYPE ptr_int Ref( int);
  OPERATORS
    ptr_int : -> ptr_int;
    ptr_int : ptr_int -> ptr_int;
ENDNEWTYPE ptr_int;EXTERNAL 'C++';
```

The example shows that information about constant arguments is lost in the SDL mapping. Also note that no difference will be made in SDL between functions that return data by value, data by reference, or constant data by reference.

Finally note that IN/OUT arguments to operators is a tool specific SDL extension.

Default Arguments

Rule: A function with default arguments are translated to several overloaded SDL operators.

This translation is reasonable given the fact that each C++ function with default arguments may be rewritten to an equivalent set of overloaded C++ functions.

Example 100: Translation of functions with default arguments

C++:

```
int func(int a, int b = 5, int c = 7);
int func(int a); // Ambiguous function!
```

SDL:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    func : int, int, int -> int;
    func : int, int -> int;
    func : int -> int;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';
```

In C++, ambiguities between overloaded functions are allowed provided that the functions are never called. SDL is, however, more strict, and

operator resolution is made from the declarations of the operators. The second version of `func` in the example above is therefore not accessible in the SDL mapping.

Unspecified Arguments

Rule: The translation of a function with unspecified arguments (a.k.a. an ellipsis function) requires the usage of an import specification that specifies the types of the unknown arguments.

See "Prototypes for Ellipsis Functions" on page 779 for information about how an import specification can be used to "expand" ellipsis functions.

Inline Functions

Rule: A function that is declared to be inline is translated as an ordinary function.

This is natural since the `inline` keyword on functions can be seen as a directive to the C++ compiler, which only affects the way that calls to these functions are generated. This is of course nothing that needs to be visible in SDL.

Example 101: Translation of inline functions

C++:

```
inline int fac(int n){/*...*/};
```

SDL:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    fac : int -> int;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';
```

Function Pointers

Rule: A function pointer is translated to an untyped pointer in SDL, i.e. to `ptr_void`.

This translation rule makes it possible to represent a function pointer in SDL, but it is not possible to call the function that it points to, or to assign the address of another function to it. That has to be done with inline C/C++ code, for example by means of the #CODE operator.

Example 102: Translation of function pointers ––––––––––––––––––

C++:

```
typedef int (*fp)(int, char);
fp g(double);
```

SDL:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    g : double -> fp;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';
SYNTYPE fp = ptr_void
ENDSYNTYPE fp;EXTERNAL 'C++';
```

––

The special `ptr_void` sort is described in "SDL Library for Fundamental C/C++ Types" on page 848.

Scope Units

Rule: A C/C++ scope unit is translated to an SDL newtype.

Note that the global scope (known as the global namespace in C++) is also translated to an SDL newtype. This newtype is called `global_namespace_`<ImpSpec> (where <ImpSpec> is the name of the import specification file) and is a container for all operators that are the translation of non-member or global functions in the program. Other global declarations are however placed directly in the SDL scope that is the context of the translation.

Example 103: Translation of the global namespace ––––––––––––––

C++:

```
int i;
void op(unsigned int);
```

SDL:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    op : unsigned_int;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';
DCL i int; EXTERNAL 'C++';
```

––

The most important scope units that may be found in a C/C++ header file are:

- Namespaces
- Classes, structs and unions
- Template classes

In C++ these scope units may be nested to arbitrary depth, but since nested newtypes are not allowed in SDL, the translation of a nested scope unit will be a newtype that has a name that is prefixed with a scope qualification prefix. This prefix consists of the names of all enclosing scope units separated by underscores ("_").

Example 104: Translation of nested scope units

C++:

```
class C {
public:
          int ci;
  class CC {
  public:
    int op();
  };
};
```

SDL:

```
NEWTYPE ptr_C_CC Ref( C_CC);
  OPERATORS
    ptr_C_CC : -> ptr_C_CC;
    ptr_C_CC : ptr_C_CC -> ptr_C_CC;
ENDNEWTYPE ptr_C_CC;EXTERNAL 'C++';
NEWTYPE C_CC /*#REFNAME 'C::CC'*/
  OPERATORS
    op : C_CC -> int;
    C_CC /*#REFNAME 'CC'*/ : -> C_CC;
    C_CC /*#REFNAME 'CC'*/ : C_CC -> C_CC;
ENDNEWTYPE C_CC;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  STRUCT
    ci int;
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

Compare the name "C_CC" of the nested class CC in this example, with the fully qualified name "C::CC" of this class in C++. The latter name is provided in a #REFNAME directive as information to the Code Generator.

Namespaces

Rule: A namespace is translated to a newtype that may not be instantiated in SDL.

Classes, structs, unions and template classes not only define scope units, but also types. They may thus be instantiated in for example variable declarations. Namespaces, on the other hand, are plain scope units and may not be instantiated. This is indicated in the SDL mapping by means of a Code Generator directive called #NOTYPE. This directive enables the Code Generator to catch attempts to instantiate newtypes that originates from namespaces.

Example 105: Translation of namespaces ––––––––––––––––––––

C++:

```
namespace N {
          const int ci;
  class CC {
  public:
    int op();
  };
  int f(char);
}
```

SDL:

```
SYNONYM N_ci /*#REFNAME 'N::ci'*/ int = EXTERNAL
'C++';
NEWTYPE ptr_N_CC Ref( N_CC);
  OPERATORS
    ptr_N_CC : -> ptr_N_CC;
    ptr_N_CC : ptr_N_CC -> ptr_N_CC;
ENDNEWTYPE ptr_N_CC;EXTERNAL 'C++';
NEWTYPE N_CC /*#REFNAME 'N::CC'*/
  OPERATORS
    op : N_CC -> int;
    N_CC /*#REFNAME 'CC'*/ : -> N_CC;
    N_CC /*#REFNAME 'CC'*/ : N_CC -> N_CC;
ENDNEWTYPE N_CC;EXTERNAL 'C++';
NEWTYPE N /*#NOTYPE*/
  OPERATORS
    N_f /*#REFNAME 'N::f'*/ : char -> int;
```

```
ENDNEWTYPE N;EXTERNAL 'C++';
```

Note that the newtype that corresponds to the namespace only contains the operators that are the translation of the functions declared in that namespace. This is analogous to how the global namespace is translated (see Example 103). Other declarations in the namespace will appear outside the newtype. All SDL declarations that are generated from namespace declarations will be prefixed with the name of the newtype that is the translation of that namespace. Also, their fully qualified C++ name is given in #REFNAME directives.

Variables

Rule: A variable is translated to an external variable if it is a non-member or global variable, or to a newtype field if it is a member variable.

Newtype fields that result from member variables of a class will be placed in the newtype that is the translation of that class. Member variables are described in "Members" on page 807, and the rest of this section will focus on non-member variables.

Example 106: Translation of non-member variables

C++:

```
int ivar, jvar;
class X {
        int j;
public:
  int Get() { return j;};
} xvar;
```

SDL:

```
DCL ivar int; EXTERNAL 'C++';
DCL jvar int; EXTERNAL 'C++';
NEWTYPE ptr_X Ref( X);
  OPERATORS
    ptr_X : -> ptr_X;
    ptr_X : ptr_X -> ptr_X;
ENDNEWTYPE ptr_X;EXTERNAL 'C++';
NEWTYPE X
  OPERATORS
    Get : X -> int;
    X : -> X;
    X : X -> X;
ENDNEWTYPE X;EXTERNAL 'C++';
```

```
DCL xvar X; EXTERNAL 'C++';
```

External variables are a tool-specific SDL extension that are similar to external synonyms.

External variables may only be declared in a process, procedure, service or operator diagram. Since CPP2SDL does not know the SDL context where the translation takes place, it has a command line option called `-novariables` that tells whether external variables may be generated or not. When CPP2SDL is executed from the Organizer, this option is set automatically. If the option is set, and a C/C++ construct is found that would map to an external variable, CPP2SDL will print a warning.

Constants

Rule: A constant is translated to an external synonym.

This rule applies for all true C/C++ constants, i.e. constants that have been declared using the `const` type specifier. It is not uncommon, especially in older C API:s, to use macros to represent constants. Such constants will not be directly accessible in SDL since the preprocessor expands them before CPP2SDL begins the translation. However, simple macro constants may often be accessed by using inline target code, for example by means of the #CODE operator. As an alternative external synonyms could be declared to represent such macros.

> **Note:**
> With the option `-extsyn` the translation of constants differs some. For constants with numeric expressions that can be calculated during translation, the default transformation rule is that also the constant value is translated. If `-extsyn` is switched on, translation is always an external synonym without its value.

Example 107Translation using -extsyn

C++:

```
const int FOO = 1;
const float ScoobieDoo = FOO/4;
const bool YOU;
```

SDL without -extsyn option (default behavior):

```
SYNONYM FOO int = 1;
SYNONYM ScoobieDoo float = 0.25;
```

```
SYNONYM YOU bool = EXTERNAL 'C++';
```

SDL with -extsyn option:

```
SYNONYM FOO int = EXTERNAL 'C++';
SYNONYM ScoobieDoo float = EXTERNAL 'C++';
SYNONYM YOU bool = EXTERNAL 'C++';
```

Example 108: Translation of constants ––––––––––––––––––––––––

C++:

```
class MyClass;
const double pi = 3.1415;
const MyClass m(7, 'x');
```

SDL:

```
SYNONYM pi double = 3.1415;
NEWTYPE MyClass /*#NOTYPE*/
ENDNEWTYPE MyClass;EXTERNAL 'C++';
SYNONYM m MyClass = EXTERNAL 'C++';
```

––

Constant Expressions

Rule: Constant expressions are evaluated while translated to SDL.

Constant expressions may be encountered at a number of places in a C/C++ header, for example as constant initializers, or as size specifiers of array declarators or bitfields. If a constant expression has to be translated to SDL, CPP2SDL attempts to evaluate it during the translation in order to simplify its representation in SDL.

Example 109: Translation of constant expressions ––––––––––––––

C++:

```
enum e {a, b, c=10};
const int i = (2+c)*b;
struct s{
  int f1 : (2+c)*b;
};
typedef int intarr[sizeof(int)+1];
```

SDL:

```
NEWTYPE e
  LITERALS a, b, c;
  OPERATORS
    IntToEnum /*#REFNAME '(e)'*/ : int -> e;
```

```
    EnumToInt : e -> int; /*#OP(PY)*/
    ORDERING;
ENDNEWTYPE e;EXTERNAL 'C++';
SYNONYM i int = 12;
NEWTYPE ptr_s Ref( s);
  OPERATORS
    ptr_s : -> ptr_s;
    ptr_s : ptr_s -> ptr_s;
ENDNEWTYPE ptr_s;EXTERNAL 'C++';
NEWTYPE s
  STRUCT
    f1 int : 12;
  OPERATORS
    s : -> s;
    s : s -> s;
ENDNEWTYPE s;EXTERNAL 'C++';
NEWTYPE arr_2_int CArray( 2, int);
ENDNEWTYPE arr_2_int;EXTERNAL 'C++';
SYNTYPE intarr = arr_2_int
ENDSYNTYPE intarr;EXTERNAL 'C++';
```

Note that not all the constant expressions in this example are visible in the SDL translation, and thus need not be evaluated by CPP2SDL.

Most constant expressions can be evaluated by CPP2SDL, but not all. In particular, expressions containing the `sizeof()` operator are difficult to evaluate since CPP2SDL has no information about what compiler that will be used to compile the generated C/C++ code. Some standard assumptions are therefore used when a `sizeof()` operator is encountered, and a warning will be issued to encourage manual inspection of the translation.

Classes, Structs and Unions

Rule: A class, struct or union is translated to an SDL newtype.

This rule follows from the fact that classes, structs and unions are scope units (see "Scope Units" on page 797).

This section mainly uses classes in the discussions and examples, but since the translation rules make no difference between classes, structs and unions, the same is valid for structs and unions. Example 110 shows the translation of an empty class, struct and union.

Example 110: Translation of classes, structs and unions ----------

C++:

```
class C {};
struct S {};
union U {};
```

SDL:

```
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
NEWTYPE ptr_S Ref( S);
  OPERATORS
    ptr_S : -> ptr_S;
    ptr_S : ptr_S -> ptr_S;
ENDNEWTYPE ptr_S;EXTERNAL 'C++';
NEWTYPE S
  OPERATORS
    S : -> S;
    S : S -> S;
ENDNEWTYPE S;EXTERNAL 'C++';
NEWTYPE ptr_U Ref( U);
  OPERATORS
    ptr_U : -> ptr_U;
    ptr_U : ptr_U -> ptr_U;
ENDNEWTYPE ptr_U;EXTERNAL 'C++';
NEWTYPE U
  OPERATORS
    U : -> U;
    U : U -> U;
ENDNEWTYPE U;EXTERNAL 'C++';
```

In the example above three C++ types translate to six SDL sorts. The reason for this is that when CPP2SDL generates a newtype for a class, it will also, by default, generate a newtype that represents a pointer type for this class. This is convenient since pointers to a class often are needed. If the class inherits other classes this pointer newtype is in fact necessary, since it then holds cast operators to the base types of the class (see "Type Compatibility between Pointers to Inherited Classes" on page 824). If the command line option `-optclasspointers` has been set, CPP2SDL will not generate this extra newtype unless a pointer to the class is explicitly present in the input code.

If the class, struct, or union has no tag, it is an incomplete type declaration. The translation rules for incomplete types are described in "Incomplete Types" on page 830.

When importing a C/C++ union, the imported PR code will contain a /*#UNIONC*/ directive for the newtype representing that union.

Anonymous Unions

Rule: An anonymous union is translated by making its members become fields of the newtype that represents the enclosing scope unit.

This translation rule is natural since an anonymous union is no scope unit.

Example 111: Translation of Anonymous Unions

C++:

```
struct S {
        int i;
  union {
    int j;
    int k;
  };
};
```

SDL:

```
NEWTYPE ptr_S Ref( S);
  OPERATORS
    ptr_S : -> ptr_S;
    ptr_S : ptr_S -> ptr_S;
ENDNEWTYPE ptr_S;EXTERNAL 'C++';
NEWTYPE S
  STRUCT
    j int;/* member of anonymous union */
    k int;/* member of anonymous union */
    i int;
  OPERATORS
    S : -> S;/*implicit parameter-less constructor*/
    S : S -> S;/* implicit copy constructor */
ENDNEWTYPE S;EXTERNAL 'C++';
```

Note that an anonymous union is not an incomplete type declaration, although the syntax is similar. An anonymous union is not used to declare a type nor a variable, and does not define a type at all. Consequently, the translation rules for anonymous unions and incomplete types differ significantly. Compare with "Incomplete Types" on page 830.

Constructors

Rule: A constructor for a class is translated to an operator with the same name as the newtype that represents the class.

The return sort of the operator will be the sort defined by the newtype for the class, and the operator will of course also be placed in that newtype.

There are two different kinds of constructors in C++:

- User-defined constructors. These constructors are manually declared and implemented.
- Implicit constructors. These constructors are implicitly declared and are auto-generated by the C++ compiler, provided that they are not already declared by the user.

While a class may contain an arbitrary number of user-defined constructors, it may at the most contain two auto-generated ones; a parameter-less (or default) constructor and a copy constructor. A parameterless constructor is available only if the class has no user-defined constructors, and a copy constructor is available only if no user-defined copy constructor is declared.

CPP2SDL will generate operators both for user-defined and implicit constructors. Example 112 below shows a class with three user-defined constructors, and one implicit copy constructor.

Example 112: Translation of constructors ––––––––––––––––––––––

C++:

```
class C {
public:
  C();
  C(int i);
  C(char c);
  ~C();
};
```

SDL:

```
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  OPERATORS
```

```
    C : -> C;
    C : char -> C;
    C : int -> C;
    C : C -> C;/* implicit copy constructor */
ENDNEWTYPE C;EXTERNAL 'C++';
```

Destructors

Rule: A destructor is not translated to SDL.

The reason why a class destructor is not made accessible in SDL, is that it normally should not be called explicitly. Instead it will be called automatically when an object of the class goes out of scope or is deleted. See Example 112 for an example of how a destructor disappears in the SDL mapping.

Members

Rule: Member variables of a C++ class are translated to fields in the newtype that is the translation of that class, and member functions are translated to operators in the same newtype.

Other declarations than variables and functions in a class, for example type declarations, are also sometimes called members of the class, but they are not translated according to the translation rule above. Instead they are considered to be declarations on their own, but defined in an enclosing scope unit (i.e. the class). See "Scope Units" on page 797 for more information.

Example 113: Translation of class members

C++:

```
class C {
public:
  int mv1; // Member variable
  void mf1(long long p1); // Member function
  enum e {a,b,c}; // "Member" type declaration
};
```

SDL:

```
NEWTYPE C_e /*#REFNAME 'C::e'*/
  LITERALS a, b, c;
  OPERATORS
    IntToEnum /*#REFNAME '(C::e)'*/ : int -> C_e;
    EnumToInt : C_e -> int; /*#OP(PY)*/
    ORDERING;
```

```
ENDNEWTYPE C_e;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  STRUCT
    mv1 int;
  OPERATORS
    mf1 : C, long_long_int;
    C : -> C;/* implicit parameter-less constructor
*/
    C : C -> C;/* implicit copy constructor */
ENDNEWTYPE C;EXTERNAL 'C++';
```

Note that the operator that represents a member function will have an additional initial formal argument. This argument has the sort of the newtype that represents the class where the member function is declared. Member functions are called from SDL in a functional style, where the first actual argument to the member function operator is the class instance on which the member function is to be invoked.

Member Access Specifier

Rule: Only members with public access specifier are translated to SDL.

This rule follows from the fact that public members of a class are the only members that are accessible from outside that class or its derived classes.

Example 114: Translation of members with different access specifiers

C++:

```
class C {
private:
        int i;
protected:
  int j;
public:
  int k;
  int GetI();
  int GetJ();
  int Calc (int x, int y);
};
```

SDL:

```
NEWTYPE ptr_C Ref( C);
```

```
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  STRUCT
    k int;
  OPERATORS
    Calc : C, int, int -> int;
    GetI : C -> int;
    GetJ : C -> int;
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

Virtual Member Functions

Rule: A virtual member function is translated in the same way as an ordinary member function.

In C++, virtual functions of a base class may be redefined in derived classes. Although this means that there is only one version of a particular virtual function in a derived class, the version defined in the base class may still be called by means of explicit qualification. Both versions of the function must thus be present in the SDL translation, exactly as is the case for non-virtual functions. See "Inheritance" on page 814 for more about how C++ inheritance is represented in SDL.

Example 115: Translation of virtual member functions

C++:

```
class CPen {
public:
  virtual void Draw(); // Virtual member function
  double GetRep(); // Non-virtual member function
};
class CPenD : public CPen {
public:
  virtual void Draw(); // Redefinition of
CPen::Draw()
};
```

SDL:

```
NEWTYPE ptr_CPen Ref( CPen);
  OPERATORS
    ptr_CPen : -> ptr_CPen;
    ptr_CPen : ptr_CPen -> ptr_CPen;
ENDNEWTYPE ptr_CPen;EXTERNAL 'C++';
```

```
NEWTYPE CPen
  OPERATORS
    Draw : CPen;
    GetRep : CPen -> double;
    CPen : -> CPen;
    CPen : CPen -> CPen;
ENDNEWTYPE CPen;EXTERNAL 'C++';
NEWTYPE ptr_CPenD Ref( CPenD);
  OPERATORS
    cast : ptr_CPenD -> ptr_CPen; /*#OP(PY)*/
    ptr_CPenD : -> ptr_CPenD;
    ptr_CPenD : ptr_CPenD -> ptr_CPenD;
ENDNEWTYPE ptr_CPenD;EXTERNAL 'C++';
NEWTYPE CPenD
  OPERATORS
    Draw : CPenD;
    CPen_Draw /*#REFNAME 'CPen::Draw'*/ : CPenD;/*
Inherited from CPen */
    GetRep : CPenD -> double;/* Inherited from CPen
*/
    CPenD : -> CPenD;/
    CPenD : CPenD -> CPenD;
ENDNEWTYPE CPenD;EXTERNAL 'C++';
```

Pure Virtual Member Functions

Rule: A pure virtual member function is translated in the same way as an ordinary member function.

Although “pure virtuality” does not affect the translation of the member function itself, it will have impact on how the containing class, which is an abstract class, is translated. The reason is that special translation rules apply for abstract classes. See “Abstract Classes” on page 826 for more information and an example on how pure virtual member functions are translated.

Static Members

Rule: A static member is translated both as an ordinary member, and as if it was declared in the global namespace.

There will thus be two representations in SDL of a static C++ member. The additional representation is caused by the fact that a static member is accessible without having an instance of the class where it is defined.

As shown in Example 116 below, a static member variable will be translated both to a newtype field and an external variable (see “Variables” on page 800), while a static member function will result in both

an operator in the newtype for the class and an operator in the special `global_namespace_ImpSpec` newtype (see "Functions" on page 793).

Example 116: Translation of static members

C++:

```
class C {
public:
  static int k;
  static void InitI(int);
};
```

SDL:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    C_InitI /*#REFNAME 'C::InitI'*/ : int;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';
DCL C_k /*#REFNAME 'C::k'*/ int; EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  STRUCT
    k int;
  OPERATORS
    InitI : C, int;
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

If the resulting SDL declarations are to be inserted in an SDL context where external variables are not allowed (i.e. if CPP2SDL executes with the `-novariables` option set), static member variables cannot be translated to external variables. In that case only the standard translation of class members can be applied. Naturally, CPP2SDL will issue a warning if this happens.

Constant Members

Rule: A constant member is translated as an ordinary member, but with a #CONSTANT directive attached.

The semantics of a constant member variable is that it may not be written to after its initialization, and a constant member function may not change the state of its object. There is no way to express these restric-

tions in SDL, so the Analyzer will not be able to detect if they are violated. However, by attaching the #CONSTANT directive to SDL declarations that result from constant members, the Code Generator can do the necessary checks.

Example 117: Translation of constant members ––––––––––––––––––

C++:

```
class C {
public:
  const int cm; // constant member
  C(int k) : cm(k) {};
  void Do(double);
                          void Undo(double) const; //
constant member function
};
```

SDL:

```
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  STRUCT
    cm /*#CONSTANT*/ int;
  OPERATORS
    C : int -> C;
    Do : C, double;
    Undo : C, double; /*#CONSTANT*/
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

––

Member Constants

Rule: A member constant is translated both as an ordinary member with a #CONSTANT directive attached, and as if it was declared in the global namespace.

This translation rule is a combination of the translation rules for static and constant members, which is natural since a member constant is declared both to be constant and static in C++.

Example 118: Translation of member constants ––––––––––––––––––

C++:

```
class X {
public:
  static const int i = 99; // member constant
};
const int X::i; // definition of i
```

SDL:

```
SYNONYM X_i /*#REFNAME 'X::i'*/ int = EXTERNAL
'C++';
NEWTYPE ptr_X Ref( X);
  OPERATORS
    ptr_X : -> ptr_X;
    ptr_X : ptr_X -> ptr_X;
ENDNEWTYPE ptr_X;EXTERNAL 'C++';
NEWTYPE X
  STRUCT
    i /*#CONSTANT*/ int;
  OPERATORS
    X : -> X;
    X : X -> X;
ENDNEWTYPE X;EXTERNAL 'C++';
```

Mutable Member Variables

Rule: A mutable member variable is translated as an ordinary member variable.

The `mutable` keyword in C++ can be looked upon as some kind of compiler directive, and needs therefore not be visible in the SDL translation.

Bitfield Member Variables

Rule: A bitfield member variable is translated to an SDL bitfield.

This rule applies for all bitfields that have a name. Bitfields without name are not translated to SDL.

It would have been possible to translate bitfields to ordinary newtype fields. However, by including the bitfield size in the SDL translation, the Analyzer is given a possibility to check that these fields are not assigned values that will not fit in the corresponding bitfield.

Example 119: Translation of bitfields ––––––––––––––––––––––––––

C++:

```
struct A {
  unsigned int i : 12;
  int : 3;
```

```
  bool dirty : 1;
};
```

SDL:

```
NEWTYPE ptr_A Ref( A);
  OPERATORS
    ptr_A : -> ptr_A;
    ptr_A : ptr_A -> ptr_A;
ENDNEWTYPE ptr_A;EXTERNAL 'C++';
NEWTYPE A
  STRUCT
    dirty bool : 1;
    i unsigned_int : 12;
  OPERATORS
    A : -> A;
    A : A -> A;
ENDNEWTYPE A;EXTERNAL 'C++';
```

Note that bitfields are a tool-specific SDL extension.

Friends

Rule: Friend declarations will not be translated to SDL.

Friendship between a class C and another declaration D only affects what members of C that the implementation of D may access. It is therefore uninteresting to supply this information in the SDL translation of the class C.

Inheritance

Rule: C++ inheritance is represented in SDL by adding the translation of all public base class members to the newtype that represents a derived class.

This rule simply means that the C++ inheritance hierarchy is flattened in the SDL newtype representation. The reason for choosing this translation strategy, instead of using SDL inheritance between newtypes, is that the semantics of C++ and SDL inheritance is quite different.

All public member variables and member functions (but not constructors) of direct or indirect bases of a class will be generated in the newtype that is the translation of that class. Such an inherited field or operator will normally have the same name in SDL as in C++, but in some cases it is necessary to prefix the name with the name of the class from which it is inherited[1]. This happens when the name of the inherited member is the same as the name of one of the members in the derived

class. Example 120 below shows how such ambiguities between inherited members are handled.

Example 120: Translation of inheritance ––––––––––––––––––––––

C++:

```
class A {
public:
  int am;
  A(char);
};
class B : public A {
public:
  char bm;
  virtual void calc();
  void set();
};
class C : public B {
public:
  int am;
  double cm;
  void calc(); // Redefines B::calc()
  void set();
};
```

SDL:

```
NEWTYPE ptr_A Ref( A);
  OPERATORS
    ptr_A : -> ptr_A;
    ptr_A : ptr_A -> ptr_A;
ENDNEWTYPE ptr_A;EXTERNAL 'C++';
NEWTYPE A
  STRUCT
    am int;
  OPERATORS
    A : char -> A;
    A : A -> A;
ENDNEWTYPE A;EXTERNAL 'C++';
NEWTYPE ptr_B Ref( B);
  OPERATORS
    cast : ptr_B -> ptr_A; /*#OP(PY)*/
    ptr_B : -> ptr_B;
    ptr_B : ptr_B -> ptr_B;
ENDNEWTYPE ptr_B;EXTERNAL 'C++';
NEWTYPE B
  STRUCT
    am int;/* Inherited from A */
    bm char;
  OPERATORS
    calc : B;
```

1. This naming rule is generalized in "Multiple Inheritance" on page 818.

```
    keyword_set /*#REFNAME 'set'*/ : B;
    B : B -> B;
ENDNEWTYPE B;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    cast : ptr_C -> ptr_A; /*#OP(PY)*/
    cast : ptr_C -> ptr_B; /*#OP(PY)*/
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  STRUCT
    am int;
    B_am /*#REFNAME 'B::am'*/ int;/* Inherited from
A */
    bm char;/* Inherited from B */
    cm double;
  OPERATORS
    calc : C;
    B_calc /*#REFNAME 'B::calc'*/ : C;/* Inherited
from B */
    keyword_set /*#REFNAME 'set'*/ : C;
    B_keyword_set /*#REFNAME 'B::keyword_set'*/ :
C;/* Inherited from B */
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

In C++, it is always possible to use a fully qualified name when accessing a class member, even if the name of the member is unambiguous without qualification. In the example above, the member variable `bm` in `C` that is inherited from `B`, may be referred to both as `bm` and `B::bm`. To avoid getting too many fields and operators in the generated newtypes, only the unqualified name can be used from SDL. This is natural since qualification in C++ normally only is done when necessary to resolve ambiguities.

There are more cases where C++ allows a member to be accessed by more than one name, while the SDL translation only supplies one of these possible names. For example, this applies for inherited types and static members as shown in Example 121 below.

Example 121: Translation of inherited types and static members ----

C++:

```
class B {
public:
        static int mv;
  static char mf(double);
  struct s {
```

```
    int y;
  } ms;
};
class D : public B {
};
```

SDL:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    B_mf /*#REFNAME 'B::mf'*/ : double -> char;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';
NEWTYPE ptr_B Ref( B);
  OPERATORS
    ptr_B : -> ptr_B;
    ptr_B : ptr_B -> ptr_B;
ENDNEWTYPE ptr_B;EXTERNAL 'C++';
NEWTYPE B
  STRUCT
    ms B_s;
    mv int;
  OPERATORS
    mf : B, double -> char;
    B : -> B;
    B : B -> B;
ENDNEWTYPE B;EXTERNAL 'C++';
DCL B_mv /*#REFNAME 'B::mv'*/ int; EXTERNAL 'C++';
NEWTYPE ptr_B_s Ref( B_s);
  OPERATORS
    ptr_B_s : -> ptr_B_s;
    ptr_B_s : ptr_B_s -> ptr_B_s;
ENDNEWTYPE ptr_B_s;EXTERNAL 'C++';
NEWTYPE B_s /*#REFNAME 'B::s'*/
  STRUCT
    y int;
  OPERATORS
    B_s /*#REFNAME 's'*/ : -> B_s;
    B_s /*#REFNAME 's'*/ : B_s -> B_s;
ENDNEWTYPE B_s;EXTERNAL 'C++';
NEWTYPE ptr_D Ref( D);
  OPERATORS
    cast : ptr_D -> ptr_B; /*#OP(PY)*/
    ptr_D : -> ptr_D;
    ptr_D : ptr_D -> ptr_D;
ENDNEWTYPE ptr_D;EXTERNAL 'C++';
NEWTYPE D
  STRUCT
    ms B_s;/* Inherited from B */
    mv int;/* Inherited from B */
  OPERATORS
    mf : D, double -> char;/* Inherited from B */
    D : -> D;
    D : D -> D;
ENDNEWTYPE D;EXTERNAL 'C++';
```

The declarations `B::mv`, `B::mf` and `B::s` in this example may in C++ also be referred to by means of the names `D::mv`, `D::mf` and `D::s`. This is not possible in the SDL translation, i.e. there are no declarations called `D_mv`, `D_mf` or `D_s`. CPP2SDL will choose the first version of the names since the members are declared in `B`.

Multiple Inheritance

The translation rule for C++ inheritance works also when a class inherits from more than one base class. However, the naming strategy described in "Inheritance" on page 814 for handling ambiguous inherited members have to be generalized to also cover the case when a class inherits the same base class more than once.

Example 122: Translation of multiple inheritance ––––––––––––––––

C++:

```
class A {
public:
  int m;
};
class B {
public:
  int m;
  int n;
};
class C: public A, public B {
};
```

SDL:

```
NEWTYPE ptr_A Ref( A);
  OPERATORS
    ptr_A : -> ptr_A;
    ptr_A : ptr_A -> ptr_A;
ENDNEWTYPE ptr_A;EXTERNAL 'C++';
NEWTYPE A
  STRUCT
    m int;
  OPERATORS
    A : -> A;
    A : A -> A;
ENDNEWTYPE A;EXTERNAL 'C++';
NEWTYPE ptr_B Ref( B);
  OPERATORS
    ptr_B : -> ptr_B;
    ptr_B : ptr_B -> ptr_B;
ENDNEWTYPE ptr_B;EXTERNAL 'C++';
NEWTYPE B
  STRUCT
```

```
    m int;
    n int;
  OPERATORS
    B : -> B;
    B : B -> B;
ENDNEWTYPE B;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    cast : ptr_C -> ptr_B; /*#OP(PY)*/
    cast : ptr_C -> ptr_A; /*#OP(PY)*/
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  STRUCT
    A_m /*#REFNAME 'A::m'*/ int;/* Inherited from A
*/
    B_m /*#REFNAME 'B::m'*/ int;/* Inherited from B
*/
    n int;/* Inherited from B */
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

The names of the generated fields and operators correspond to the qualified names that must be used in C++ to access the members in question. The rule is to qualify an ambiguous member with the most specialized base class that makes the name of the member unique. In most cases this base class is the class where the ambiguous member is declared, but when the inheritance hierarchy forms a graph rather than a tree (see Example 123) it might be necessary to qualify with the name of a class further down on the inheritance path.

Note that in some extraordinary inheritance hierarchies, it is possible that a member of a base class is inaccessible in a derived class. This happens when the inherited member cannot be unambiguously qualified according to the naming rule described above. If this happens, CPP2SDL will not translate the member to SDL, and a warning will be printed.

Virtual and Non-Virtual Inheritance

Rule: Virtual inheritance is translated in the same way as ordinary inheritance.

Virtual inheritance affects the way data is replicated when multiple inheritance is used. As shown in “Multiple Inheritance” on page 818

members that are inherited more than once from the same base class need to be prefixed in SDL.

Since data from a base class is not replicated in derived classes that inherit from the base class with virtual inheritance, it would be possible to avoid prefixing the name of the members that are virtually inherited from the base class. However, since a virtually inherited member in general may be accessed using many alternative prefixes (corresponding to possible paths for reaching the member in the inheritance graph), and none of these prefixes can be said to be more natural to use than the others, all versions of the member's name are included in the SDL translation. This is the reason why no difference is made between virtual and non-virtual inheritance in SDL.

Example 123: Translation of virtual inheritance ––––––––––––––––––

C++:

```
class A {
public:
  int a;
};
class C : public A {
};
class D : public virtual A {
};
class E : public virtual A {
};
class G : public C, public D, public E {
};
```

SDL:

```
NEWTYPE ptr_A Ref( A);
  OPERATORS
    ptr_A : -> ptr_A;
    ptr_A : ptr_A -> ptr_A;
ENDNEWTYPE ptr_A;EXTERNAL 'C++';
NEWTYPE A
  STRUCT
    a int;
  OPERATORS
    A : -> A;
    A : A -> A;
ENDNEWTYPE A;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    cast : ptr_C -> ptr_A; /*#OP(PY)*/
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
```

```
NEWTYPE C
  STRUCT
    a int;/* Inherited from A */
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
NEWTYPE ptr_D Ref( D);
  OPERATORS
    cast : ptr_D -> ptr_A; /*#OP(PY)*/
    ptr_D : -> ptr_D;
    ptr_D : ptr_D -> ptr_D;
ENDNEWTYPE ptr_D;EXTERNAL 'C++';
NEWTYPE D
  STRUCT
    a int;/* Inherited from A */
  OPERATORS
    D : -> D;
    D : D -> D;
ENDNEWTYPE D;EXTERNAL 'C++';
NEWTYPE ptr_E Ref( E);
  OPERATORS
    cast : ptr_E -> ptr_A; /*#OP(PY)*/
    ptr_E : -> ptr_E;
    ptr_E : ptr_E -> ptr_E;
ENDNEWTYPE ptr_E;EXTERNAL 'C++';
NEWTYPE E
  STRUCT
    a int;/* Inherited from A */
  OPERATORS
    E : -> E;
    E : E -> E;
ENDNEWTYPE E;EXTERNAL 'C++';
NEWTYPE ptr_G Ref( G);
  OPERATORS
    cast : ptr_G -> ptr_E; /*#OP(PY)*/
    cast : ptr_G -> ptr_D; /*#OP(PY)*/
    cast : ptr_G -> ptr_C; /*#OP(PY)*/
    ptr_G : -> ptr_G;
    ptr_G : ptr_G -> ptr_G;
ENDNEWTYPE ptr_G;EXTERNAL 'C++';
NEWTYPE G
  STRUCT
    C_a /*#REFNAME 'C::a'*/ int;/* Inherited from A
*/
    D_a /*#REFNAME 'D::a'*/ int;/* Inherited from A
*/
    E_a /*#REFNAME 'E::a'*/ int;/* Inherited from A
*/
  OPERATORS
    G : -> G;
    G : G -> G;
ENDNEWTYPE G;EXTERNAL 'C++';
```

Inheritance Access Specifier

Rule: Only members that are inherited using public inheritance are translated to SDL.

When the inheritance is private or protected, the members of the base class are not accessible from outside the class. It is therefore natural to exclude members, that are inherited from private and protected bases, in the newtype that represents the derived class.

Example 124: Translation of inheritance with different access specifiers ––

C++:

```
class X {
public:
  int a;
  void f();
};
class Y1 : public X {};
class Y2 : protected X {};
class Y3 : private X {};
```

SDL:

```
NEWTYPE ptr_X Ref( X);
  OPERATORS
    ptr_X : -> ptr_X;
    ptr_X : ptr_X -> ptr_X;
ENDNEWTYPE ptr_X;EXTERNAL 'C++';
NEWTYPE X
  STRUCT
    a int;
  OPERATORS
    f : X;
    X : -> X;
    X : X -> X;
ENDNEWTYPE X;EXTERNAL 'C++';
NEWTYPE ptr_Y1 Ref( Y1);
  OPERATORS
    cast : ptr_Y1 -> ptr_X; /*#OP(PY)*/
    ptr_Y1 : -> ptr_Y1;
    ptr_Y1 : ptr_Y1 -> ptr_Y1;
ENDNEWTYPE ptr_Y1;EXTERNAL 'C++';
NEWTYPE Y1
  STRUCT
    a int;/* Inherited from X */
  OPERATORS
    f : Y1;/* Inherited from X */
    Y1 : -> Y1;
    Y1 : Y1 -> Y1;
ENDNEWTYPE Y1;EXTERNAL 'C++';
```

```
NEWTYPE ptr_Y2 Ref( Y2);
  OPERATORS
    ptr_Y2 : -> ptr_Y2;
    ptr_Y2 : ptr_Y2 -> ptr_Y2;
ENDNEWTYPE ptr_Y2;EXTERNAL 'C++';
NEWTYPE Y2
  OPERATORS
    Y2 : -> Y2;
    Y2 : Y2 -> Y2;
ENDNEWTYPE Y2;EXTERNAL 'C++';
NEWTYPE ptr_Y3 Ref( Y3);
  OPERATORS
    ptr_Y3 : -> ptr_Y3;
    ptr_Y3 : ptr_Y3 -> ptr_Y3;
ENDNEWTYPE ptr_Y3;EXTERNAL 'C++';
NEWTYPE Y3
  OPERATORS
    Y3 : -> Y3;
    Y3 : Y3 -> Y3;
ENDNEWTYPE Y3;EXTERNAL 'C++';
```

The inheritance access specifier also affects how casting from the derived type to the base type can be done. This is described in "Type Compatibility between Inherited Classes" on page 823 and in "Type Compatibility between Pointers to Inherited Classes" on page 824.

Type Compatibility between Inherited Classes

Rule: An object of a derived class may be assigned to an object of a base class by using an explicit cast operator in SDL.

The above assignment (known as slicing) is type-compatible in C++ without the use of a cast operator. Since only the common members are copied in the assignment, this operation is somewhat dangerous and is not generally recommended. Therefore, the cast operators that are needed in SDL to do slicing between objects, are only generated when the `-slicing` option is set.

Example 125: Generation of operators for slicing----------

C++:

```
class C {};
class D {};
class CD : public C, public D {};
```

SDL:

```
NEWTYPE ptr_C Ref( C);
  OPERATORS
```

```
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
NEWTYPE ptr_D Ref( D);
  OPERATORS
    ptr_D : -> ptr_D;
    ptr_D : ptr_D -> ptr_D;
ENDNEWTYPE ptr_D;EXTERNAL 'C++';
NEWTYPE D
  OPERATORS
    D : -> D;
    D : D -> D;
ENDNEWTYPE D;EXTERNAL 'C++';
NEWTYPE ptr_CD Ref( CD);
  OPERATORS
    cast : ptr_CD -> ptr_D; /*#OP(PY)*/
    cast : ptr_CD -> ptr_C; /*#OP(PY)*/
    ptr_CD : -> ptr_CD;
    ptr_CD : ptr_CD -> ptr_CD;
ENDNEWTYPE ptr_CD;EXTERNAL 'C++';
NEWTYPE CD
  OPERATORS
    cast : CD -> D; /*#OP(PY)*/
    cast : CD -> C; /*#OP(PY)*/
    CD : -> CD;
    CD : CD -> CD;
ENDNEWTYPE CD;EXTERNAL 'C++';
```

Note that the inheritance access specifier affects how these cast operators are generated. A cast operator from a class `D` to a class `B` will only be generated if `B` is a public unambiguous base of `D`. If it is private or protected, or is an ambiguous base for `D`, it is not allowed to cast from `D` to `B`.

Type Compatibility between Pointers to Inherited Classes

Rule: A pointer to an object of a derived class may be assigned to a pointer to an object of a base class by using an explicit cast operator in SDL.

The above assignment is type-compatible in C++, i.e. “up-casts” in a class hierarchy are implicit in C++. This is an important property of object-oriented languages that support for example polymorphism. In SDL, however, the newtypes for a base class and a derived class will be unrelated and thus type incompatible. To support up-casting in SDL,

explicit cast operators are generated in the newtype that represents the pointer type to a derived class.

Example 126: Generation of cast operators for up-casting ––––––––––

C++:

```
class C {};
class D {};
class CD : public C, public D {};
```

SDL:

```
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
NEWTYPE ptr_D Ref( D);
  OPERATORS
    ptr_D : -> ptr_D;
    ptr_D : ptr_D -> ptr_D;
ENDNEWTYPE ptr_D;EXTERNAL 'C++';
NEWTYPE D
  OPERATORS
    D : -> D;
    D : D -> D;
ENDNEWTYPE D;EXTERNAL 'C++';
NEWTYPE ptr_CD Ref( CD);
  OPERATORS
    cast : ptr_CD -> ptr_D; /*#OP(PY)*/
    cast : ptr_CD -> ptr_C; /*#OP(PY)*/
    ptr_CD : -> ptr_CD;
    ptr_CD : ptr_CD -> ptr_CD;
ENDNEWTYPE ptr_CD;EXTERNAL 'C++';
NEWTYPE CD
  OPERATORS
    CD : -> CD;
    CD : CD -> CD;
ENDNEWTYPE CD;EXTERNAL 'C++';
```

––

Note that the inheritance access specifier is taken into consideration so that a cast operator from `Ref(D)` to `Ref(B)` only will be generated if the class `B` is a public unambiguous base of the class `D`.

Sometimes it is necessary to do down-casts, or even cross-casts, in a class hierarchy. Such casts (known as dynamic casts) are explicit both

in C++ and SDL. See "Run-Time Type Information and Dynamic Cast" on page 827 for more information.

Abstract Classes

Rule: An abstract class is translated to a newtype without constructor operators.

This translation rule makes it possible to declare pointers to an abstract class, but no objects of such a class may be allocated since there are no constructor operators that can be used as argument to the `new` operator (see "Dynamic Memory Management" on page 832).

Example 127: Translation of abstract classes ––––––––––––––––––––

C++:

```
class C {
public:
  virtual int f(int) = 0; // pure virtual member
function
  C() {};
};
class D : public C {
};
```

SDL:

```
NEWTYPE ptr_C Ref( C);
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  OPERATORS
    f : C, int -> int;
ENDNEWTYPE C;EXTERNAL 'C++';
NEWTYPE ptr_D Ref( D);
  OPERATORS
    cast : ptr_D -> ptr_C; /*#OP(PY)*/
ENDNEWTYPE ptr_D;EXTERNAL 'C++';
NEWTYPE D
  OPERATORS
    f : D, int -> int;/* Inherited from C */
ENDNEWTYPE D;EXTERNAL 'C++';
```

––

Note from the example above that abstractness is inherited to a derived class if not each pure virtual member function of all its base classes are redefined in the derived class.

Run-Time Type Information and Dynamic Cast

Rule: A pointer to an object of a base class may be assigned to a pointer to an object of a derived class, or to a pointer to an object of a base class of such a derived class, by using an explicit cast operator in SDL.

The above assignments require a dynamic cast in C++, which is done with an explicit cast operator that supports down-casts and cross-casts in an inheritance hierarchy. Since these casts require run-time type information (RTTI) about the dynamic type of an object, most C++ compilers have an option that must be set to safely support dynamic casts. For the same reason, CPP2SDL also has such an option called `-rtti`. If it is set, cast operators will be generated that enable the type conversions that are possible in C++ using dynamic casts.

The source type of a dynamic cast must be polymorphic, i.e. contain one or more virtual member functions, possibly inherited ones. For each such polymorphic class `X`, cast operators will be generated that convert from `Ref(X)` to `Ref(Y)`, for each class `Y` that either inherits from `X` (down-casts), or is a public base class of a class that inherits from `X` (cross-casts).

Example 128 below illustrates this translation rule. It is assumed that all classes in the example contain virtual functions and thus are polymorphic.

Example 128: Generation of cast operators for dynamic casting ––––

C++:

```
class A {};
class B: public A {};
class E {};
class D: protected E {};
class C: public B, public D {};
```

SDL:

```
NEWTYPE ptr_A Ref( A);
  OPERATORS
    ptr_A : -> ptr_A;
    ptr_A : ptr_A -> ptr_A;
ENDNEWTYPE ptr_A;EXTERNAL 'C++';
NEWTYPE A
  OPERATORS
    A : -> A;
    A : A -> A;
ENDNEWTYPE A;EXTERNAL 'C++';
NEWTYPE ptr_B Ref( B);
```

```
  OPERATORS
    cast /*#REFNAME 'dynamic_cast<B*>'*/ : ptr_A ->
ptr_B;
    cast : ptr_B -> ptr_A; /*#OP(PY)*/
    ptr_B : -> ptr_B;
    ptr_B : ptr_B -> ptr_B;
ENDNEWTYPE ptr_B;EXTERNAL 'C++';
NEWTYPE B
  OPERATORS
    B : -> B;
    B : B -> B;
ENDNEWTYPE B;EXTERNAL 'C++';
NEWTYPE ptr_E Ref( E);
  OPERATORS
    ptr_E : -> ptr_E;
    ptr_E : ptr_E -> ptr_E;
ENDNEWTYPE ptr_E;EXTERNAL 'C++';
NEWTYPE E
  OPERATORS
    E : -> E;
    E : E -> E;
ENDNEWTYPE E;EXTERNAL 'C++';
NEWTYPE ptr_D Ref( D);
  OPERATORS
    ptr_D : -> ptr_D;
    ptr_D : ptr_D -> ptr_D;
ENDNEWTYPE ptr_D;EXTERNAL 'C++';
NEWTYPE D
  OPERATORS
    D : -> D;
    D : D -> D;
ENDNEWTYPE D;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    cast /*#REFNAME 'dynamic_cast<A*>'*/ : ptr_D ->
ptr_A;
    cast /*#REFNAME 'dynamic_cast<B*>'*/ : ptr_D ->
ptr_B;
    cast /*#REFNAME 'dynamic_cast<D*>'*/ : ptr_A ->
ptr_D;
    cast /*#REFNAME 'dynamic_cast<D*>'*/ : ptr_B ->
ptr_D;
    cast /*#REFNAME 'dynamic_cast<C*>'*/ : ptr_D ->
ptr_C;
    cast : ptr_C -> ptr_D; /*#OP(PY)*/
    cast /*#REFNAME 'dynamic_cast<C*>'*/ : ptr_A ->
ptr_C;
    cast : ptr_C -> ptr_A; /*#OP(PY)*/
    cast /*#REFNAME 'dynamic_cast<C*>'*/ : ptr_B ->
ptr_C;
    cast : ptr_C -> ptr_B; /*#OP(PY)*/
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
```

```
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

Note that no cast operators are generated that cast to `ptr_E` since `E` is a protected base of `D`. But in fact there are no cast operators that cast from `ptr_E` neither. The somewhat subtle reason for this is that those operators cannot be used in practice, since the protected inheritance makes it impossible to have a variable with `ptr_E` as static type and `ptr_C` as dynamic type. CPP2SDL will therefore not generate these cast operators.

In C++ dynamic casts work both for pointers and references to objects. In SDL, however, it is only possible to do dynamic casts between pointers, since references are not explicitly represented in the translation (see "References" on page 789).

Forward Declarations

Rule: A forward declaration is not translated to SDL.

This rule is valid for all forward declarations for which there are definitions later on in the header file. This is the most common case, and the purpose of such forward declarations is simply to make an identifier known to the C/C++ compiler so that it may be used before it is defined.

However, it is possible to make a forward declaration for which no definition exists in the header file. In that case CPP2SDL must generate an extra newtype to represent the missing definition. Since this extra newtype does not correspond to a real C/C++ type, it is marked with a #NOTYPE directive.

Example 129 below contains two forward declarations, one of which has no corresponding definition (`class C`).

Example 129: Translation of forward declarations ————————————

C++:

```
typedef struct S *fwdS;
class C *fwdC;
struct S {
        int a;
};
```

SDL:

```
SYNTYPE fwdS = ptr_S
ENDSYNTYPE fwdS;EXTERNAL 'C++';
NEWTYPE C /*#NOTYPE*/
ENDNEWTYPE C;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
DCL fwdC ptr_C; EXTERNAL 'C++';
NEWTYPE ptr_S Ref( S);
  OPERATORS
    ptr_S : -> ptr_S;
    ptr_S : ptr_S -> ptr_S;
ENDNEWTYPE ptr_S;EXTERNAL 'C++';
NEWTYPE S
  STRUCT
    a int;
  OPERATORS
    S : -> S;
    S : S -> S;
ENDNEWTYPE S;EXTERNAL 'C++';
```

Incomplete Types

Rule: An incomplete type declaration is translated to a newtype that may not be instantiated in SDL. This rule applies to all incomplete types even if they are declared within complete types. For example a tagless type within a container type will not be correctly instantiated in SDL.

Compare this translation rule with the one for namespaces (see "Namespaces" on page 799). While a namespace does not define a type at all, an incomplete type declaration defines an incomplete type that may not be referred to. That is the reason why such a newtype must not be instantiated in SDL.

C/C++ allows declarations of incomplete classes, structs, unions and enums, i.e. all types having a tag. Incomplete types are therefore also called tag-less types.

Incomplete types can be used in

- data declarations (i.e. variables, constants etc.)
- type declarations (i.e. typedefs)
- "useless" declarations (i.e. without declaring data or type)

Example 130: Translation of incomplete types ––––––––––––––––––

C++:

```
typedef struct {
  int i;
} r;
struct S {
  int i;
  struct {
    int j;
  } ss1, *ss2, ss3[2]; // Data declarations
};
typedef enum {
  a, b, c
} ss1, *ss2, ss3[2]; // Type declarations
typedef struct {
        int i;
}; // Missing type name - "useless" declaration
struct {
        int i;
}; // Missing variable name - "useless" declaration
```

SDL:

```
NEWTYPE r
  STRUCT
    i int;
ENDNEWTYPE r;EXTERNAL 'C++';
NEWTYPE S_incomplete_ss3
  STRUCT
    j int;
ENDNEWTYPE S_incomplete_ss3;
NEWTYPE ptr_S_incomplete_ss3 Ref( S_incomplete_ss3);
  OPERATORS
    ptr_S_incomplete_ss3 : -> ptr_S_incomplete_ss3;
    ptr_S_incomplete_ss3 : ptr_S_incomplete_ss3 ->
    ptr_S_incomplete_ss3;
ENDNEWTYPE ptr_S_incomplete_ss3;EXTERNAL 'C++';
NEWTYPE arr_2_S_incomplete_ss3 CArray(2,
S_incomplete_ss3);
ENDNEWTYPE arr_2_S_incomplete_ss3;EXTERNAL 'C++';
NEWTYPE ptr_S Ref( S);
  OPERATORS
    ptr_S : -> ptr_S;
    ptr_S : ptr_S -> ptr_S;
ENDNEWTYPE ptr_S;EXTERNAL 'C++';
NEWTYPE S
  STRUCT
    i int;
    ss1 S_incomplete_ss3;
    ss2 ptr_S_incomplete_ss3;
    ss3 arr_2_S_incomplete_ss3;
  OPERATORS
    S : -> S;
```

```
    S : S -> S;
ENDNEWTYPE S;EXTERNAL 'C++';
NEWTYPE incomplete_ss3
  LITERALS a, b, c;
  OPERATORS
    EnumToInt : incomplete_ss3 -> int; /*#OP(PY)*/
    ORDERING;
ENDNEWTYPE incomplete_ss3;EXTERNAL 'C++';
NEWTYPE ptr_incomplete_ss3 Ref( incomplete_ss3);
  OPERATORS
    ptr_incomplete_ss3 : -> ptr_incomplete_ss3;
    ptr_incomplete_ss3 : ptr_incomplete_ss3 ->
    ptr_incomplete_ss3;

ENDNEWTYPE ptr_incomplete_ss3;EXTERNAL 'C++';
SYNTYPE ss2 = ptr_incomplete_ss3
ENDSYNTYPE ss2;EXTERNAL 'C++';
NEWTYPE arr_2_incomplete_ss3 CArray( 2,
incomplete_ss3);
ENDNEWTYPE arr_2_incomplete_ss3;EXTERNAL 'C++';
SYNTYPE ss3 = arr_2_incomplete_ss3
ENDSYNTYPE ss3;EXTERNAL 'C++';
```

As can be seen in the example, incomplete types that are used in data or type declarations will have the name of the last declared variable or type, prefixed with a user-configurable string that by default is “incomplete_”. The option `-prefix` can be used to configure this string.

Incomplete types in “useless” declarations will not be translated to SDL, and CPP2SDL will issue warnings that the declarations were ignored.

Note that the translation of incomplete enum declarations differs from the normal translation rule of an enum declaration. The differences are listed in “Enumerated Types” on page 790.

Finally note that incomplete classes, structs and unions define scope units although they are incomplete. Their names thus follows the rules for naming of scope units described in “Scope Units” on page 797.

Dynamic Memory Management

Rule: The C/C++ primitives for dynamic memory management is represented in SDL by means of special operators.

Dynamic memory management is done differently in C and C++. How C or C++ data is dynamically allocated and deallocated in SDL there-

fore depends on whether CPP2SDL executes in C or C++ mode (controlled by the option `-c`). In both cases dynamic memory management is done by means of special SDL operators that are defined in the `Ref` generator. However, the definition of the `Ref` generator is different in C and C++ mode (see "SDL Library for Fundamental C/C++ Types" on page 848).

C Mode

The following operators are used to support dynamic memory management of C data from SDL:

- `Make!`
  Enables dynamic allocation of simple C data.
- `free`
  Enables dynamic deallocation of data that was allocated by the `Make!` operator.

Example 131: Dynamic memory management of C data from SDL ---

C:

```
struct S {
  int i;
  double j;
};
```

SDL:

```
NEWTYPE ptr_S Ref( S);
ENDNEWTYPE ptr_S;EXTERNAL 'C';
NEWTYPE S /*#REFNAME 'struct S'*/
  STRUCT
    i int;
    j double;
ENDNEWTYPE S;EXTERNAL 'C';
```

SDL Usage:

```
dcl var s, ptrs ptr_s;
task {
  ptrs := (. var .);
  ptrs*>!i := 4;
  free(ptrs);
};
```

C++ Mode

The following operators are used to support dynamic memory management of C++ data from SDL:

- `new`[1]
  Enables dynamic allocation of scalar C++ data of, for example, class type, fundamental type, or pointer type. It corresponds to the C++ operator with the same name.

- `delete`
  Enables dynamic deallocation of data that was allocated by the `new` operator. It corresponds to the C++ operator with the same name.

- `new_array`
  Enables dynamic allocation of arrays of C++ data of, for example, class type, fundamental type, or pointer type. It corresponds to the C++ operator `new[]`.

- `delete_array`
  Enables dynamic deallocation of data that was allocated by the `new_array` operator. It corresponds to the C++ operator `delete[]`.

Example 132: Dynamic memory management of C++ data from SDL –

C++:

```
struct S {
  int i;
  double j;
};
```

Import Specification:

```
TRANSLATE {
  S**
  int*
}
```

SDL:

```
NEWTYPE ptr_int Ref( int);
  OPERATORS
    ptr_int : -> ptr_int;
    ptr_int : ptr_int -> ptr_int;
ENDNEWTYPE ptr_int;EXTERNAL 'C++';
NEWTYPE ptr_ptr_S Ref( ptr_S);
```

1. In fact the `Make!` operator can also be used in C++ mode. In that case it behaves exactly like the `new` operator.

```
  OPERATORS
    ptr_ptr_S : -> ptr_ptr_S;
    ptr_ptr_S : ptr_ptr_S -> ptr_ptr_S;
ENDNEWTYPE ptr_ptr_S;EXTERNAL 'C++';
NEWTYPE ptr_S Ref( S);
  OPERATORS
    ptr_S : -> ptr_S;
    ptr_S : ptr_S -> ptr_S;'
ENDNEWTYPE ptr_S;EXTERNAL 'C++';
NEWTYPE S
  STRUCT
    i int;
    j double;
 OPERATORS
    S : -> S;
    S : S -> S;
ENDNEWTYPE S;EXTERNAL 'C++';
```

SDL Usage:

```
dcl ptrs ptr_s, ptrptrs ptr_ptr_s, ptri ptr_int;
task {
  ptrs := new(s);
  ptrs*>!i := 4;
  ptrptrs := new(ptr_s);
  ptri := new(int);
  delete(ptrs);
  delete(ptrptrs);
  delete(ptri);
  ptri := newArray(int, 5);
  deleteArray(ptri);
};
```

The input to the `new` and `new_array` operators must be an operator that corresponds to a constructor in C++. To enable dynamic allocation of data with non-class types, there must thus exist constructor operators for these types. These operators correspond to the implicit parameter-less and copy constructors, which exist for each C++ type. The definition of these constructor operators are part of the non-generated SDL declarations that are included when the `-generatecpptypes` option is set (see "SDL Library for Fundamental C/C++ Types" on page 848).

Overloaded Operators

Rule: An overloaded C++ operator is translated to a corresponding overloaded SDL operator.

Since the sets of operators that may be overloaded are different in C++ and SDL, not all overloaded C++ operators can be translated to SDL.

The table below shows which C++ operators that can be translated to SDL

:

C++ operator	Description	SDL operator
+	(binary) Addition	+
-	(binary) Subtraction	-
*	(binary) Multiplication	*
*	(unary prefix) Dereference	*>
/	(binary) Division	/
%	(binary) Modulo	rem
!	(unary prefix) Not	not
<	(binary) Less	<
>	(binary) Greater	>
<<	(binary) Left Shift	<
>>	(binary) Right Shift	>
==	(binary) Equal	=
!=	(binary) Not Equal	/=
<=	(binary) Less Equal	<=
>=	(binary) Greater Equal	>=
&&	(binary) And	and
||	(binary) Or	or

Note that even if there is a corresponding SDL operator to translate to, a C++ operator could be declared in a way that would make it necessary to qualify its SDL name, which is not possible. This happens for example if the operator is declared to be static, or declared inside a namespace (see "Scope Units" on page 797 for more about scope name qualifications). It may also happen due to name qualification rules for inherited members (see "Multiple Inheritance" on page 818). CPP2SDL will issue a warning if it encounters an overloaded operator that cannot be translated.

The table above shows that the translation of the less and greater operators (<, >) is the same as the translation of the shift operators (<<, >>). Obviously, this may lead to ambiguities if both these operator pairs are overloaded in a class. In that case, the less and greater operators will have precedence, and CPP2SDL will issue a warning that the overloaded shift operators cannot be translated to SDL.

Example 133: Translation of overloaded operators ––––––––––––––

C++:

```
class ostream {
public:
  ostream& operator<(const char* p1);
  ostream& operator<<(const char* p1);
  ostream& operator>>(const char* p1);
  static int operator%(int p1);
  bool operator!();
};
```

SDL:

```
NEWTYPE ptr_char Ref( char);
  OPERATORS
    ptr_char : -> ptr_char;
    ptr_char : ptr_char -> ptr_char;
ENDNEWTYPE ptr_char;EXTERNAL 'C++';
NEWTYPE ptr_ostream Ref( ostream);
  OPERATORS
    ptr_ostream : -> ptr_ostream;
    ptr_ostream : ptr_ostream -> ptr_ostream;
ENDNEWTYPE ptr_ostream;EXTERNAL 'C++';
NEWTYPE ostream
  OPERATORS
    "not" /*#REFNAME 'operator!'*/ : ostream ->
bool;
    "rem" /*#REFNAME 'operator%'*/ : ostream, int ->
int;
    "<" /*#REFNAME 'operator<'*/ : ostream, ptr_char
-> ostream;
    ostream : -> ostream;
    ostream : ostream -> ostream;
ENDNEWTYPE ostream;EXTERNAL 'C++';
```

Conversion Operators

Rule: A conversion operator is translated to a special `conv` operator in SDL.

In C++, a conversion operator is implicitly called by the compiler when a matching type conversion is needed. The `conv` operator, however, must be called explicitly in SDL.

Example 134: Translation of conversion operators

C++:

```
class Tiny {
public:
  operator int(); // Implicit conversion from Tiny
to int
};
```

SDL:

```
NEWTYPE ptr_Tiny Ref( Tiny);
  OPERATORS
    ptr_Tiny : -> ptr_Tiny;
    ptr_Tiny : ptr_Tiny -> ptr_Tiny;
ENDNEWTYPE ptr_Tiny;EXTERNAL 'C++';
NEWTYPE Tiny
  OPERATORS
    conv : Tiny -> int; /*#OP(PY)*/
    Tiny : -> Tiny;
    Tiny : Tiny -> Tiny;
ENDNEWTYPE Tiny;EXTERNAL 'C++';
```

Note that the `conv` operator returns the target type of the type conversion specified by the conversion operator. The #OP(PY) directive tells the Code Generator that the operator is implicitly called in C++.

Templates

Rule: A template declaration is translated to SDL by instantiating it.

A template declaration as such cannot be translated to SDL. Only specified instantiations of the template can be translated. CPP2SDL will print a warning about this when a template declaration is encountered.

A template instantiation may of course be present in the input headers. In that case CPP2SDL will translate the template instantiation by substituting all formal template arguments in the template declaration with the actual template arguments used in the template instantiation. If the input headers contain no suitable instantiation of a certain template, an import specification may be used to provide such an instantiation. See "Template Instantiations" on page 780 to learn more about that.

There are two main kinds of template declarations in C++; class templates and function templates.

Class Templates

Rule: An instantiation of a class template is translated in the same way as the non-template class that is obtained if all formal arguments of the class template declaration are substituted with the actual arguments of the class template instantiation.

This translation rule implies that class template instantiations will become newtypes in SDL. The name of such a newtype will consist of the name of the template class, followed by the names of all actual template arguments of the template instantiation. The name will also be prefixed with a string that by default is “tpl_”. The option `-prefix` can be used to configure this string.

Example 135: Translation of class template instantiations ––––––––

C++:

```
template <class T> class C {
public:
        T t;
  T f();
  C(T v);
};
typedef C<int> mytype; // Class template
instantiation
```

SDL:

```
NEWTYPE ptr_tpl_C_int Ref( tpl_C_int);
  OPERATORS
    ptr_tpl_C_int : -> ptr_tpl_C_int;
    ptr_tpl_C_int : ptr_tpl_C_int -> ptr_tpl_C_int;
ENDNEWTYPE ptr_tpl_C_int;EXTERNAL 'C++';
NEWTYPE tpl_C_int /*#REFNAME 'C<int >'*/
  STRUCT
    t int;
  OPERATORS
    tpl_C_int /*#REFNAME 'C'*/ : int -> tpl_C_int;
    f : tpl_C_int -> int;
    tpl_C_int /*#REFNAME 'C'*/ : tpl_C_int ->
tpl_C_int;
ENDNEWTYPE tpl_C_int;EXTERNAL 'C++';
SYNTYPE mytype = tpl_C_int
ENDSYNTYPE mytype;EXTERNAL 'C++';
```

––

Note that a #REFNAME directive passes the C++ name of the class template instantiation to the Code Generator.

Function Templates

Rule: An instantiation of a function template is translated in the same way as the non-template function that is obtained if all formal arguments of the function template declaration are substituted with the actual arguments of the function template instantiation.

This translation rule implies that function template instantiations will become operators in SDL. The name of such an operator will consist of the name of the template function, followed by the names of all actual template arguments of the template instantiation. The name will also be prefixed with a string that by default is “tpl_”. The option `-prefix` can be used to configure this string.

Since a function template is instantiated when called, a C++ header file will normally not contain any function template instantiations. Instead an import specification should be used to provide the necessary instantiations (see “Template Instantiations” on page 780). In Example 136 below an import specification is used to instantiate the template function with the type `int*`.

Example 136: Translation of function template instantiations ––––––––

C++:

```
template <class T> T func(T t);
```

Import Specification:

```
TRANSLATE {
  func<int*>
}
```

SDL:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    tpl_func_ptr_int /*#REFNAME 'func<int* >'*/ :
ptr_int
-> ptr_int;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';
NEWTYPE ptr_int Ref( int);
  OPERATORS
    ptr_int : -> ptr_int;
    ptr_int : ptr_int -> ptr_int;
```

```
ENDNEWTYPE ptr_int;EXTERNAL 'C++';
```

As can be seen from the translation of the example above, the #REFNAME directive contains the C++ name of the template instantiation written on the so called explicit form[1]. This makes the Code Generator use this explicit form when the template function is called from SDL.

> **Note:**
>
> Calling a function template from SDL requires that the target C++ compiler can handle calls using the explicit form of the function template instantiation.

Default Template Arguments

Rule: An instantiation of a template declaration with default arguments is translated in the same way as an ordinary template instantiation, where omitted actual arguments in the instantiation are substituted with the specified default types or values.

This translation rule is very similar to the one used for functions with default arguments (see "Default Arguments" on page 795).

Example 137: Translation of templates with default arguments

C++:

```
template <class T, class U = char, int i = 5> class
C {
public:
        T t[i];
  T f();
  C(U p1);
};
C<int> var1; // Using all the default values
C<int, bool> var2; // Using the default value for i
C<int, bool, 5> var3; // Not using any default value
```

SDL:

```
NEWTYPE ptr_tpl_C_int_char_5 Ref( tpl_C_int_char_5);
  OPERATORS
```

1. In the explicit form of a function template instantiation, all actual template arguments are provided explicitly in the instantiation rather than being deduced from the types of the actual arguments in a call to the function template.

```
    ptr_tpl_C_int_char_5 : -> ptr_tpl_C_int_char_5;
    ptr_tpl_C_int_char_5 : ptr_tpl_C_int_char_5 ->
    ptr_tpl_C_int_char_5;
ENDNEWTYPE ptr_tpl_C_int_char_5;EXTERNAL 'C++';
NEWTYPE tpl_C_int_char_5 /*#REFNAME 'C<int, char, 5
>'*/
  STRUCT
    t arr_5_int;
  OPERATORS
    tpl_C_int_char_5 /*#REFNAME 'C'*/ : char ->
tpl_C_int_char_5;
    f : tpl_C_int_char_5 -> int;
    tpl_C_int_char_5 /*#REFNAME 'C'*/ :
tpl_C_int_char_5
-> tpl_C_int_char_5;
ENDNEWTYPE tpl_C_int_char_5;EXTERNAL 'C++';
DCL var1 tpl_C_int_char_5; EXTERNAL 'C++';
DCL var2 tpl_C_int_bool_5; EXTERNAL 'C++';
NEWTYPE arr_5_int CArray( 5, int);
ENDNEWTYPE arr_5_int;EXTERNAL 'C++';
NEWTYPE ptr_tpl_C_int_bool_5 Ref( tpl_C_int_bool_5);
  OPERATORS
    ptr_tpl_C_int_bool_5 : -> ptr_tpl_C_int_bool_5;
    ptr_tpl_C_int_bool_5 : ptr_tpl_C_int_bool_5 ->
    ptr_tpl_C_int_bool_5;
ENDNEWTYPE ptr_tpl_C_int_bool_5;EXTERNAL 'C++';
NEWTYPE tpl_C_int_bool_5 /*#REFNAME 'C<int, bool, 5
>'*/
  STRUCT
    t arr_5_int;
  OPERATORS
    tpl_C_int_bool_5 /*#REFNAME 'C'*/ : bool ->
tpl_C_int_bool_5;
    f : tpl_C_int_bool_5 -> int;
-> tpl_C_int_bool_5;
ENDNEWTYPE tpl_C_int_bool_5;EXTERNAL 'C++';
DCL var3 tpl_C_int_bool_5; EXTERNAL 'C++';
```

Note that although the template instantiations of `var2` and `var3` in the example above look different, they evaluate to the same template type both in C++ and in the SDL translation.

Miscellaneous

This section covers some miscellaneous issues that have not been discussed so far. They are divided into constructs that are part of the C or C++ languages, and constructs that are not part of the languages as such, but that nevertheless may be found in an input C/C++ header file.

Language Constructs

Volatile

Rule: A volatile declaration is translated in the same way as an ordinary declaration.

The `volatile` specifier can be looked upon as some kind of compiler directive, and needs therefore not be visible in the SDL translation.

Linkage

Rule: The linkage of a C/C++ identifier is not visible in the SDL translation of the identifier.

There is one important exception to this rule; static linkage of class members affects their translation as described in "Static Members" on page 810.

In general, the linkage of a C/C++ identifier can be specified to be internal or external using the keywords `static` or `extern` (although the former is a deprecated feature in C++ for all declarations but class members). In C++ it is also possible to use the `extern` keyword to specify that a set of declarations have C linkage, i.e. belong to a translation unit that is compiled with a C compiler.

Example 138: Translation of identifiers with different linkage–––––––

C++:

```
extern int a; // Declaration of a
extern int a; // Legal redeclaration of a
int a; // Definition of a
extern "C" {
  struct S {
    int x;
  };
}
```

SDL:

```
DCL a int; EXTERNAL 'C++';
NEWTYPE ptr_S Ref( S);
  OPERATORS
    ptr_S : -> ptr_S;
    ptr_S : ptr_S -> ptr_S;
ENDNEWTYPE ptr_S;EXTERNAL 'C++';
NEWTYPE S
  STRUCT
    x int;
```

```
  OPERATORS
    S : -> S;
    S : S -> S;
ENDNEWTYPE S;EXTERNAL 'C++';
```

Note that the `extern "C"` directive in this example does not affect the mapping of `S` at all. For example, it will be possible to instantiate `S` using the `new` operator.

Non-Language Constructs

Macros

Rule: Macros are not translated to SDL.

The reason for not translating macros is that they are not part of the C or C++ languages. Macros are expanded and removed by the preprocessor before CPP2SDL performs the translation.

Some header files (especially C headers) contain numerous macros that could be useful or even essential to access in SDL. Fortunately most macros can actually be accessed from SDL, although they are not translated by CPP2SDL. Refer to "Constants" on page 801 for more information.

SDL Sorts in C/C++

Rule: A C/C++ type called "SDL_<sort>", where <sort> is a predefined SDL sort, is translated to that SDL sort.

Since this translation rule restricts the way ordinary C/C++ types may be named, it is only respected by CPP2SDL if the `-sdlsorts` option is set.

Example 139: Translation of types referring to SDL sorts

C++:

```
SDL_Real func(SDL_Integer, int);
```

SDL:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    func : Integer, int -> Real;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';
```

The feature of referring to SDL sorts from a C/C++ header file may be useful if the header has been designed to be used from SDL exclusively.

Special Translation Rules for C Compilers

CPP2SDL by default assumes that its input is C++ code, and that the target compiler is a C++ compiler. In order to also support C code and C target compilers, the translation rules have to be slightly modified. CPP2SDL does this if the `-c` option is set. The tool then executes in “C mode”.

A general difference is how the external specifier for all SDL declarations will be generated. Normally this specifier is followed by the string ‘C++’ to tell the Code Generator that the declaration was translated from a C++ declaration. In C mode the string will instead be ‘C’.

Example 140: Different external specifier in C mode ––––––––––––

C++:

```
const int a;
```

SDL:

```
SYNONYM a int = EXTERNAL 'C';
```

––

In C, a struct or a union is not a scope unit, which means that declarations inside a struct or a union should be treated as ordinary declarations. This means that the #REFNAME directive that normally is used for specifying the qualified name of for example a nested struct declaration, will not be printed in C mode.

Instead, a #REFNAME directive will be inserted after the name of newtypes that represent tagged types (i.e. structs, unions and enums). The reason is that C, contrary to C++, does not allow such types to be referenced only with the name of the tag. Another place where the “full” type name is required in C mode is in the C style cast that is generated by means of a #REFNAME directive for `IntToEnum` operators (see “Enumerated Types” on page 790).

Example 141: Differences in translation of structs, unions and enums

C++:

```
struct S {
  int a;
  struct SS {
    int b;
  };
};
```

```
typedef enum E {e1, e2, e3} Etype;
```

SDL:

```
NEWTYPE ptr_SS Ref( SS);
ENDNEWTYPE ptr_SS;EXTERNAL 'C';
NEWTYPE SS /*#REFNAME 'struct SS'*/
  STRUCT
    b int;
ENDNEWTYPE SS;EXTERNAL 'C';
NEWTYPE ptr_S Ref( S);
ENDNEWTYPE ptr_S;EXTERNAL 'C';
NEWTYPE S /*#REFNAME 'struct S'*/
  STRUCT
    a int;
ENDNEWTYPE S;EXTERNAL 'C';
NEWTYPE E /*#REFNAME 'enum E'*/
  LITERALS e1, e2, e3;
  OPERATORS
    IntToEnum /*#REFNAME '(enum E)'*/ : int -> E;
    EnumToInt : E -> int; /*#OP(PY)*/
    ORDERING;
ENDNEWTYPE E;EXTERNAL 'C';
SYNTYPE Etype = E
ENDSYNTYPE Etype;EXTERNAL 'C';
```

Finally, note that memory allocation is done differently in C and C++. This is reflected in SDL by using a different definition of the `Ref` generator when CPP2SDL executes in C mode, where for example the `new`, `delete`, `new_array`, and `delete_array` operators are not present. See "Dynamic Memory Management" on page 832 and "SDL Library for Fundamental C/C++ Types" on page 848 for more information.

SDL Library for Fundamental C/C++ Types

The SDL declarations that are generated by CPP2SDL will normally not be semantically correct on their own. They typically contain several references to SDL sorts that represent fundamental C/C++ types, for example `int`, `char` and `bool`, and type declarators such as pointers (`*`) and arrays (`[]`). The SDL representations of all fundamental C/C++ types and type declarators are defined in a library consisting of a few SDL/PR files. The table below lists these files and their contents.

SDL/PR file	Contents
BasicCTypes.pr *Contains SDL representations of fundamental C types. Also contains representations for an untyped pointer (void*) and the array type declarator ([]).*	SYNTYPE int = Integer ENDSYNTYPE int; SYNTYPE unsigned_int = Integer ENDSYNTYPE unsigned_int; SYNTYPE long_int = Integer ENDSYNTYPE long_int; SYNTYPE unsigned_long_int = Integer ENDSYNTYPE unsigned_long_int; SYNTYPE short_int = Integer ENDSYNTYPE short_int; SYNTYPE unsigned_short_int = Integer ENDSYNTYPE unsigned_short_int; SYNTYPE char = Character ENDSYNTYPE char; SYNTYPE signed_char = Character ENDSYNTYPE signed_char; SYNTYPE unsigned_char = Octet ENDSYNTYPE unsigned_char; SYNTYPE float = Real ENDSYNTYPE float; SYNTYPE double = Real ENDSYNTYPE double; NEWTYPE ptr_void LITERALS Null; DEFAULT Null; ENDNEWTYPE ptr_void; GENERATOR CArray (CONSTANT Length, TYPE Itemsort) OPERATORS modify!: CArray, Integer, Itemsort -> CArray; extract!: CArray, Integer -> Itemsort; ENDGENERATOR CArray;

SDL/PR file	Contents
BasicC++Types.pr *Contains SDL representations of fundamental C++ types. Also contains operators representing implicit constructors for fundamental types.* *Note that this file includes the files BasicCtypes.pr and ExtraCTypes.pr.*	/*#INCLUDE 'BasicCTypes.pr'*/ /*#INCLUDE 'ExtraCTypes.pr'*/ SYNTYPE bool = Boolean ENDSYNTYPE bool; NEWTYPE wchar_t ENDNEWTYPE wchar_t; NEWTYPE __ConstructorOperators /*#NOTYPE*/ OPERATORS int: -> int; int: int -> int; unsigned_int: -> unsigned_int; unsigned_int: unsigned_int -> unsigned_int; long_int: -> long_int; long_int: long_int -> long_int; unsigned_long_int: -> unsigned_long_int; unsigned_long_int: unsigned_long_int -> unsigned_long_int; short_int: -> short_int; short_int: short_int -> short_int; unsigned_short_int: -> unsigned_short_int; unsigned_short_int: unsigned_short_int -> unsigned_short_int; char: -> char; char: char -> char; signed_char: -> signed_char; signed_char: signed_char -> signed_char; unsigned_char: -> unsigned_char; unsigned_char: unsigned_char -> unsigned_char; float: -> float; float: float -> float; double: -> double; double: double -> double; ptr_void: -> ptr_void; ptr_void: ptr_void -> ptr_void; bool: -> bool; bool: bool -> bool; wchar_t: -> wchar_t; wchar_t: wchar_t -> wchar_t; ENDNEWTYPE __ConstructorOperators;EXTERNAL 'C++';

SDL/PR file	Contents
ExtraCTypes.pr *Contains SDL representations of additional fundamental C types.*	`SYNTYPE long_long_int = Integer` `ENDSYNTYPE long_long_int;` `SYNTYPE unsigned_long_long_int = Integer` `ENDSYNTYPE unsigned_long_long_int;`
ExtraC++Types.pr *Contains SDL representations of additional fundamental C++ types.*	`NEWTYPE__ExtraConstructorOperators` `long_long_int : -> long_long_int;` `long_long_int : long_long_int -> long_long_int;` `unsigned_long_long_int : ->` `unsigned_long_long_int;` `unsigned_long_long_int : unsigned_long_long_int -` `> unsigned_long_long_int` `ENDNEWTYPE__ExtraConstructorOperators`
CPointer.pr *Contains SDL representation of the C pointer type declarator (*).*	`GENERATOR Ref (TYPE Itemsort)` `LITERALS Null, Alloc;` `OPERATORS` `  modify!  : Ref, Integer, Itemsort -> Ref;` `  extract! : Ref, Integer -> Itemsort;` `  "*>"     : Ref, Itemsort -> Ref;` `  "*>"     : Ref -> Itemsort;` `  "&"      : Itemsort -> Ref;` `  make!    : Itemsort -> Ref;` `  free     : in/out Ref;` `  "+"      : Ref, Integer -> Ref;` `  "-"      : Ref, Integer -> Ref;` `  cast     : Ref -> ptr_void;` `  cast     : ptr_void -> Ref;` `DEFAULT Null;` `ENDGENERATOR Ref;`

SDL/PR file	Contents
C++Pointer.pr *Contains SDL representation of the C++ pointer type declarator (*).*	GENERATOR Ref (TYPE Itemsort) LITERALS Null; OPERATORS modify! : Ref, Integer, Itemsort -> Ref; extract! : Ref, Integer -> Itemsort; "*>" : Ref, Itemsort -> Ref; "*>" : Ref -> Itemsort; "&" : Itemsort -> Ref; new : Itemsort -> Ref; delete : Ref; new_array : Itemsort, Integer -> Ref; delete_array : Ref; "+" : Ref, Integer -> Ref; "-" : Ref, Integer -> Ref; cast : Ref -> ptr_void; cast : ptr_void -> Ref; DEFAULT Null; ENDGENERATOR Ref;

If the option `-generatecpptypes` is set, CPP2SDL will include some of the files from the table above in the SDL translation. Which files that are included depends on if CPP2SDL executes in C or C++ mode (controlled by the `-c` option).

The following files will be included in C mode:

- `BasicCTypes.pr`
- `CPointer.pr`

The following files will be included in C++ mode:

- `BasicC++Types.pr`
- `C++Pointer`

The reason for breaking out the types `long long int` and `unsigned long long int` into separate files, is that not all compilers support these types. These files must be manually included if these types are present in the input headers.

Hint:

The syntype definitions of the SDL sorts that represent fundamental C/C++ types can easily be changed. For example, the definitions of the SDL sorts ‘char’ and ‘unsigned char’ could be swapped if the target platform specifies that a simple ‘char’ is unsigned rather than signed.

Example usage of some C/C++ functionality

Overloaded Operators

This example illustrates how to call an operator which has been made accessible by CPP2SDL. There are two operators defined and used, the first being a member operator, and the second a non-member operator. Also see “Overloaded Operators” on page 853.

C++:

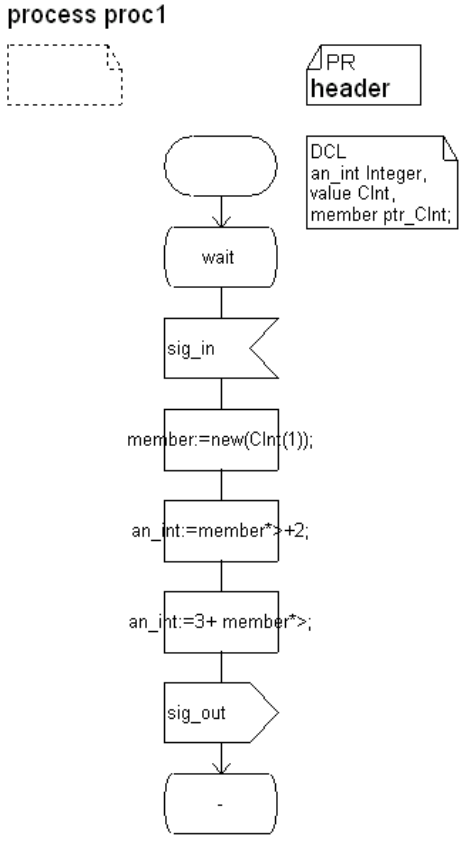
```
class CInt {
  int val;
        public:
  CInt() : val(0) {};
  CInt(int i) : val(i) {};
  int value() const { return val; };

  int operator+(const int& i){val+= i; return val;};
};

int operator+(const int& left, const CInt& right)
{return right.value()+left;};
```

SDL:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    "+" /*#REFNAME 'operator+'*/ : int, CInt -> int;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';
NEWTYPE ptr_CInt Ref( CInt);
  OPERATORS
    ptr_CInt : -> ptr_CInt;
    ptr_CInt : ptr_CInt -> ptr_CInt;
ENDNEWTYPE ptr_CInt;EXTERNAL 'C++';
NEWTYPE CInt
  OPERATORS
    "+" /*#REFNAME 'operator+'*/ : CInt, int -> int;
    CInt : -> CInt;
    CInt : int -> CInt;
```

```
    value : CInt -> int; /*#CONSTANT*/
    CInt : CInt -> CInt;
ENDNEWTYPE CInt;EXTERNAL 'C++';
```

Use in SDL:



String handling

It is possible to do C-style string handling in SDL, by using the standard C header **`string.h`**. By including '`string.h`' and '`stdio.h`' we are given access to the functions defined within them in SDL. You may notice that strcpy is defined in the hand written header as well as in '`string.h`'. The former definition allows us to assign the string "good-

bye" to empty, without using the return value of strcpy, and importing it to SDL. Also needed is an allocate and deallocate function. An example allocate function has been defined in the header, a deallocate function should also be done in the practice to avoid memory leaks.

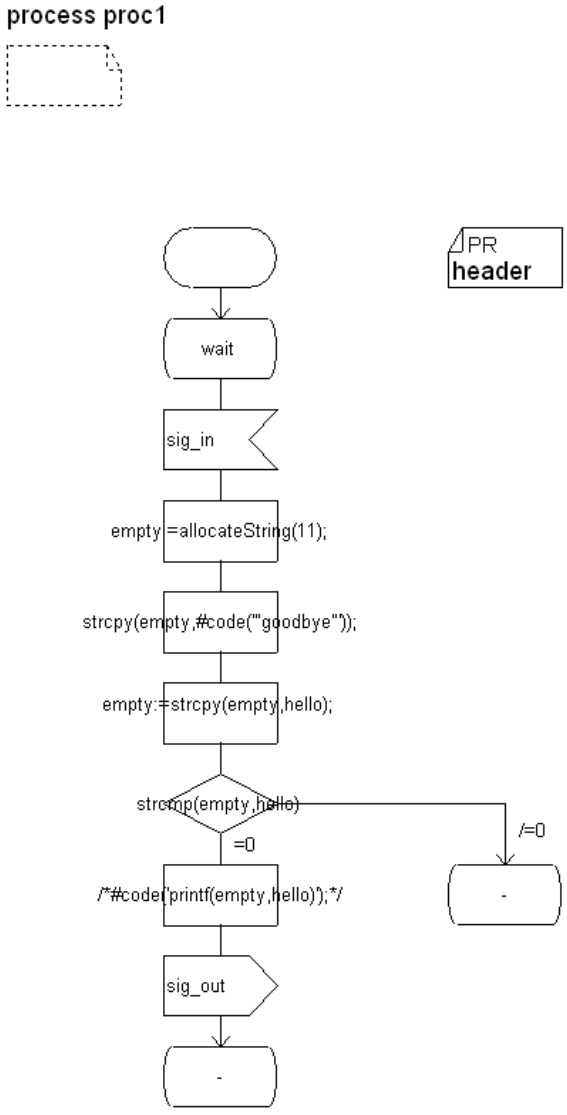
C:

```
#include<string.h>
#include<stdio.h>

#ifdef __CPP2SDL__
void strcpy(char*,char*);
#endif

typedef char* string;
char ara[10];
string hello= "hello";
string empty;

char* allocateString(int length) {
  return (char*) calloc(length,sizeof(char));
}
```

SDL:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    memcpy : ptr_void, ptr_void, size_t -> ptr_void;
    memcmp : ptr_void, ptr_void, size_t -> int;
    memset : ptr_void, int, size_t -> ptr_void;
    _strset : ptr_char, int -> ptr_char;
    strcpy : ptr_char, ptr_char -> ptr_char;
    strcat : ptr_char, ptr_char -> ptr_char;
    strcmp : ptr_char, ptr_char -> int;
    strlen : ptr_char -> size_t;
    ....
    unlink : ptr_char -> int;
    strcpy : ptr_char, ptr_char;
    allocateString : int -> ptr_char;
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C';
...
SYNTYPE string = ptr_char
ENDSYNTYPE string;EXTERNAL 'C';
/*#SDTREF(TEXT,header_CPP2SDL.i,728,6)*/
NEWTYPE arr_10_char CArray( 10, char);
ENDNEWTYPE arr_10_char;EXTERNAL 'C';
/*#SDTREF(TEXT,header_CPP2SDL.i,728,6)*/
DCL ara arr_10_char; EXTERNAL 'C';
/*#SDTREF(TEXT,header_CPP2SDL.i,729,8)*/
DCL hello string; EXTERNAL 'C';
/*#SDTREF(TEXT,header_CPP2SDL.i,730,8)*/
DCL empty string; EXTERNAL 'C';
```

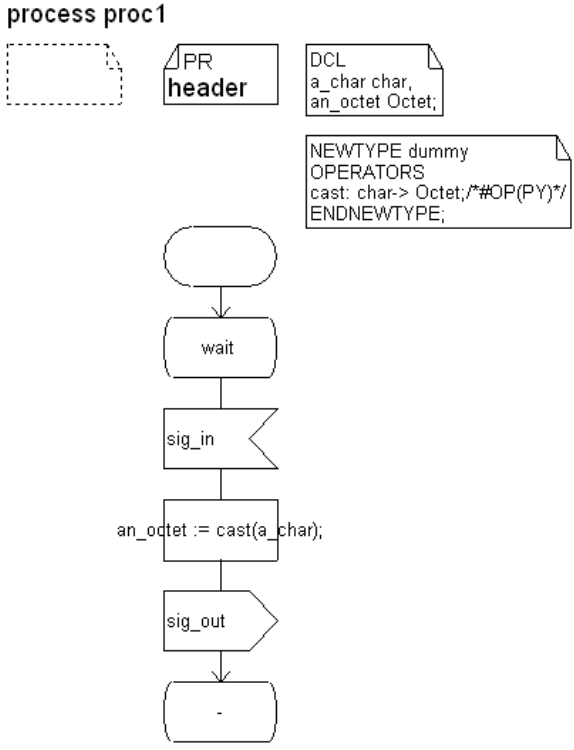
Use in SDL:


process proc1
PR
header
wait
sig_in
empty:=allocateString(11);
strcpy(empty,#code("goodbye"));
empty:=strcpy(empty,hello);
strcmp(empty,hello)
=0
/=0
/*#code("printf(empty,hello)");*/
-
sig_out
-

Type conversion

Some type conversions are easier in C/C++ than in SDL, in particular those that are implicit. An implicit type conversion must often be explicit in SDL, by introducing a simple cast operator that performs the conversion. For example, by using the mapping of SDL sorts between unsigned_char and char we can create a cast operator in SDL that converts a char to an Octet. (See "Fundamental Types" on page 786 for more information). This corresponds to the implicit conversion in C/C++ between char and unsigned char.

Use in SDL:



Error Handling

The input headers to CPP2SDL have in many cases been compiled with a C/C++ compiler previously, and it should then be relatively uncommon that CPP2SDL will have to issue any error messages. However, differences in language support, and inappropriate preprocessor settings, are common sources for error reports also from input files that otherwise are perfectly correct.

If CPP2SDL finds an error in the input, a message will be printed that briefly describes the reason for the error.

> **Note:**
>
> The error messages produced by CPP2SDL are often less descriptive than the corresponding error messages from a C/C++ compiler. If CPP2SDL reports errors in a header file, it is therefore a good idea to run a C/C++ compiler on the same header file to get more information about the reason for the error.

The format of printed error messages are described in "Source and Error References" on page 783.

CPP2SDL performs a complete syntactic analysis of the input C/C++ code, and syntax errors are reported as shown in Example 142 below.

Example 142: Syntax errors ----------

File syn.h:

```
int f(};
conts int i = 7;
```

Command Prompt:

```
% cpp2sdl syn.h
Parsing C/C++ input...
Syntax errors found. Cannot perform SDL translation.
#SDTREF(TEXT,syn.h,1,7)
ERROR 3200 Syntax error.
#SDTREF(TEXT,syn.h,2,7)
ERROR 3200 Syntax error.
2 errors and 0 warnings.
```

CPP2SDL will proceed with semantic analysis only if no errors were found during the syntactic analysis. The semantic analysis that is done

by CPP2SDL is not complete according to the C/C++ standards, but only a limited number of semantic tests are performed:

- The type of variables, constants, typedefs, functions, function arguments, actual template arguments, and base classes are checked. If a type is undeclared, or if it depends on an undeclared type[1], an error message will be issued.
- The actual arguments of a template instantiation are checked against the formal arguments of the template definition. If there are too few or too many actual arguments, or if there are type mismatches between actual and formal arguments, an error message will be issued.

Example 143: Semantic errors ––––––––––––––––––––––––––––––

File sem.h:

```
template <class U, int d> class S {
public:
  U arr[d];
};
typedef unknown T; // T depends on undeclared type
const unknown a = 3;
T f(int, char);
S<T, 3> var;
typedef S<> t1; // Too few actual arguments
typedef S<char, 5, 5> t2; // Too many actual
arguments
typedef S<int, int> t3; // Argument type mismatch
```

Command Prompt:

```
% cpp2sdl -errorlimit 10 sem.h
Parsing C/C++ input...
Translating C/C++ to SDL...
Generating SDL...
#SDTREF(TEXT,sem.h,11,9)
ERROR 3263 Illegal instantiation of template 'S'.
#SDTREF(TEXT,sem.h,10,9)
ERROR 3263 Illegal instantiation of template 'S'.
#SDTREF(TEXT,sem.h,9,9)
ERROR 3263 Illegal instantiation of template 'S'.
#SDTREF(TEXT,sem.h,8,3)
ERROR 3261 The type 'T' is undeclared, or is
depending on an undeclared type.
#SDTREF(TEXT,sem.h,7,3)
ERROR 3261 The type 'T' is undeclared, or is
```

1. One example of such a type dependency is when the source type of a typedef type is undeclared. Usages of the typedef type will then be considered to be undeclared.

```
depending on an undeclared type.
#SDTREF(TEXT,sem.h,6,15)
ERROR 3261 The type 'unknown' is undeclared, or is
depending on an undeclared type.
#SDTREF(TEXT,sem.h,5,17)
ERROR 3261 The type 'unknown' is undeclared, or is
depending on an undeclared type.
#SDTREF(TEXT,sem.h,1,33)
WARNING 3211 Cannot translate template declaration.
The declaration will be ignored.
7 errors and 1 warnings.
```

Note that the command line option `-errorlimit` can be used to set a limit for the number of errors to report before terminating a translation.

CPP2SDL Messages

CPP2SDL may produce three kinds of messages during the translation of a set of header files.

- Error messages are printed if CPP2SDL finds any syntactic or semantic errors in the input header files. See "Example usage of some C/C++ functionality" on page 853 for more information about how CPP2SDL handles errors in the input.
- Warnings are given if CPP2SDL finds language constructs that for some reason cannot be fully translated. The tool also prints warnings if it has to make assumptions about a construct that not necessarily are valid.
- Information messages are all other messages that are printed.

The rest of this section lists and explains all errors and warnings that may be issued by CPP2SDL.

Errors

`ERROR 3200 Syntax error.`

An error was found during the syntactic analysis of the input. CPP2SDL will not continue with semantic analysis and translation to SDL, since the program is not correct.

> **Note:**
> If this error message is printed for a program that is accepted by a C/C++ compiler, make sure that the correct language dialect has been set to CPP2SDL by means of the `-dialect` option.

`ERROR 3260 The identifier <identifier name> is undeclared.`

An identifier is undeclared, i.e. the program is not semantically correct and will therefore not be translated to SDL.

`ERROR 3261 The type <type name> is undeclared, or is depending on an undeclared type.`

A type is undeclared, or depends on a type that is undeclared. A type defined by a typedef of an undeclared type is an example of a type that depends on a undeclared type. Since the program is not semantically correct, it will not be translated to SDL.

```
ERROR 3262 The base <base class name> is undeclared.
```

A class inherits from a base class that is undeclared. This is a semantic error, and the program will thus not be translated to SDL.

```
ERROR 3263 Illegal instantiation of template <template
name>.
```

A template instantiation is semantically incorrect. Make sure that the number of actual arguments in the template instantiation matches the number of formal arguments in the template declaration, and that the kinds of the arguments are correct. Since the program is not semantically correct, it will not be translated to SDL.

Warnings

```
WARNING 3201 Static member variable <variable name> will
not be globally accessible since no SDL variables are al-
lowed.
```

A static member variable cannot be fully translated, since no external variables are allowed in the context where the generated SDL declarations are to be injected. The static variable will still be accessible as an ordinary member variable, but not as a globally accessible variable. Unset the option `-novariables` to allow generation of external SDL variables.

```
WARNING 3202 Static overloaded operator <operator name>
will not be globally accessible.
```

A static overloaded operator cannot be fully translated, since it is not possible to qualify the name of an overloaded operator in SDL which otherwise would be required. The static operator will still be accessible as an ordinary member overloaded operator, but not as a globally accessible overloaded operator. Refer to "Overloaded Operators" on page 835.

```
WARNING 3203 Cannot translate incomplete type declara-
tion without declared objects. The declaration will be
ignored.
```

An incomplete type declaration that is not used as the type of at least one object (e.g. variable, constant, or type) is a useless declaration that will not be translated to SDL. See "Incomplete Types" on page 830 for more about useless incomplete type declarations.

```
WARNING 3204 Cannot translate overloaded operator, since
it is declared in a namespace.
```

An overloaded operator declared in a namespace cannot be translated to SDL, since it is not possible to qualify the name of an overloaded operator in SDL which otherwise would be required. Refer to "Overloaded Operators" on page 835.

```
WARNING 3205 Cannot translate overloaded shift operator,
since the '<' or '>' operator also is overloaded in this
scope.
```

The translation rule for overloaded operators only supports translation of either the < and > operators or the << and >> operators. This warning is given if an operator from both these operator pairs are overloaded in a certain scope. See "Overloaded Operators" on page 835 for more information.

```
WARNING 3206 Cannot translate overloaded operator, since
no corresponding SDL operator exists.
```

An overloaded operator cannot be translated to SDL, since no appropriate SDL operator exists that could represent it. The table in "Overloaded Operators" on page 835 shows what overloaded C++ operators that may be represented in SDL.

```
WARNING 3207 Unable to evaluate sizeof expression prop-
erly.
```

A constant expression contains a usage of the `sizeof()` operator, and could therefore not be safely evaluated by CPP2SDL. The translation of the constant expression may thus be incorrect, and should be manually reviewed. See "Constant Expressions" on page 802 for more information about constant expressions.

```
WARNING 3208 The member <member name> of <class name> in-
herited via <base class names> is inaccessible and will
not be translated.
```

An inherited class member cannot be accessed in C++ due to a combination of multiple inheritance and base classes with members having the same name. The member will thus not be translated to SDL.

```
WARNING 3209 Cannot translate function pointer type. It
will be represented by ptr_void.
```

This warning is given when a function pointer type is encountered in the input. The support for function pointers is limited (see "Function Pointers" on page 796), and they will be represented as untyped pointers in SDL (i.e. `ptr_void`).

```
WARNING 3210 Cannot translate typedef of function type.
The declaration will be ignored.
```

A typedef declaration where the source type is a function type cannot be translated to SDL, since there is no translation rule for function types.

```
WARNING 3211 Cannot translate template declaration. The
declaration will be ignored.
```

A template declaration cannot be translated to SDL. Only instantiations of a template declaration can be translated. Note that this warning is given also when a template instantiation has been specified in an import specification (see "Template Instantiations" on page 780). In that case the warning could be ignored.

```
WARNING 3212 Cannot fully translate ellipsis function.
Unspecified function arguments will be ignored.
```

A function with unspecified arguments (a.k.a. an ellipsis function) cannot be fully translated to SDL, since no information has been provided about the unspecified arguments. The function will be translated, but without taking the unspecified arguments into consideration. See "Prototypes for Ellipsis Functions" on page 779 to learn how to use an import specification to provide CPP2SDL with actual arguments for unspecified formal arguments of ellipsis functions.

```
WARNING 3213 The typedef name <name> conflicts with the
name of another non-compatible type. The declaration
will be ignored.
```

A typedef declaration cannot be translated to SDL, since the typedef name is the same as another type that is not type compatible with the type defined by the typedef itself. This warning may be given for typedefs of pointers or arrays of tagged types. For example:

```
typedef struct T {
  int i;
} *T;
```

This declaration, which is illegal in C++ but legal in C, contains two types called `T` that are type incompatible. CPP2SDL will make the type

`struct T` available in SDL (called `T` there), while the type `T` will not be translated.

It is recommended to change the name of either the typedef name or the type tag to enable CPP2SDL to translate both types, and thus avoid getting this warning.

```
WARNING 3290 The identifier <identifier name> does not
refer to a declared object. It will be ignored.
```

This warning is given if CPP2SDL finds an identifier in an import specification that does not exist in the input program. The identifier will be ignored.

```
WARNING 3291 Cannot translate the identifier <identifier
name>, since it is a class member.
```

This warning is given if CPP2SDL finds an identifier in an import specification that refers to a class member in the input program. The identifier will not be translated, since only declarations in namespaces may be translated (see "Import Specifications" on page 776).

Chapter 14 The CPP2SDL Tool

Chapter

15 *CPP2SDL Migration Guide*

This chapter is a practical guide on how to convert an SDL specification that uses the H2SDL tool, which is not supported any longer, into an equivalent specification that uses the new CPP2SDL tool.

A complete description of the CPP2SDL tool can be found in "The CPP2SDL Tool" on page 761 in chapter 14, *The CPP2SDL Tool* .

Introduction

The purpose of this chapter is to help H2SDL users to smoothly migrate from the obsolete H2SDL tool to CPP2SDL. In addition, IBM Rational Customer Support will be happy to assist in solving any migration problems not covered by this chapter.

The H2SDL tool provided automated support for accessing C code from within the SDL Suite. The CPP2SDL tool works according to the same fundamental principle, i.e. by automatically translating C/C++ declarations into SDL representations.

However, due to the difference between the C and C++ languages, CPP2SDL is not entirely backward compatible with H2SDL. This means that existing SDL systems that use H2SDL, may need to be updated when upgrading to CPP2SDL.

Reasons to Migrate

Apart from H2SDL being obsolete, there are several reasons to migrate from H2SDL to CPP2SDL:

- CPP2SDL has superior language support. Not only does it support C++, but it also covers a larger set of the C language.
- CPP2SDL allows the user to configure the translation from C/C++ to SDL by means of import specifications. This is a technique which often cuts the build time significantly, when large APIs are interfaced from SDL.
- With H2SDL, the user often has to edit input header files, to make them suitable for translation. This need is significantly reduced with CPP2SDL, since the import specification mechanism lets the user configure the translation of each C/C++ declaration individually.
- Translation rules implemented by CPP2SDL are considerably improved, compared to the ones implemented by H2SDL. These improvements are due to the implementation of language extensions oriented towards SDL2000.
- CPP2SDL has an improved reference generator, allowing more precise analysis.
- The SDL package generated by H2SDL may only be used at system level, while the SDL declarations generated by CPP2SDL may be injected at any level in the SDL hierarchy. This is possible by means of a new SDL-PR symbol in the SDL Editor.
- CPP2SDL supports a wider range of preprocessors and compilers to be used for preprocessing input header files.
- CPP2SDL is built with state-of-the-art compiler technology on top of a commercial C/C++ parser. This provides for rapid incorporation of future extensions to C/C++.

Migration Guidelines

This section describes the procedure of upgrading an SDL system from H2SDL to CPP2SDL usage. Required changes are grouped into subsections corresponding to the main reasons for the changes.

Note:

The CPP2SDL tool takes advantage of SDL2000 being case-sensitive, therefore you should make sure that your SDL system is possible to analyze in case-sensitive mode prior to starting the migration work.

The SDL Suite includes a tool that helps you convert your SDL system into a case-sensitive SDL system (see Update to case-sensitive SDL.

Update to case-sensitive SDL

The following steps will help you convert your SDL system into case-sensitive mode.

1. Make sure you use the standard Text Editor for editing text files in the SDL Suite.
2. Files should be write enabled. If not, the files will be updated but stored with the extension `.keep`.
3. Check the correctness of the system with the batch commands described in "Checking diagrams for duplicated object IDs" on page 212 in chapter 2, *The Organizer*.

   You should now be ready to convert to case-sensitive mode. Start with creating an extended cross reference file and a file with references to all keywords in the system.
4. Issue the following command:

   ```
   sdtbatch[.bat] -a systemPath/systemname.sdt -options
   optionFile.txt
   ```

   The file `optionFile.txt` should include the following lines of options to the Analyzer:

   ```
   [ANALYSEROPTIONS]
   SDLKeywordFile=True
   SetPredefinedXRef=True
   XRef=True
   CaseSensitiveSDL=False
   ```

   Two files will be created in the target directory of the system. They are called `casesensitive.xrf` and `casesensitive.key` and should be used in the next step.

5. Now you can start the update of the system by invoking the following command.

   ```
   sdtbatch[.bat] -changecase
   targetDir/casesensitive.xrf
   ```

   The result of the operation is sent to `sdtout` and contains information about what have been done, problems that have arisen and finally which files that was updated and saved. Since it is a lot of information you should probably pipe the output to a file.

   When the case is updated we recommend that you analyze the system again. Proceed as follow:

6. Change the last row in the `optionFile.txt` from above. The last row should now be:

   ```
   CaseSensitiveSDL=True
   ```

7. If your system uses the ctypes package and H2SDL you will need to use a special ctypes package that makes it possible to analyze in case-sensitive mode. Open the system in the Organizer and connect ctypes package to the file `<installation directory>/sdt/include/ADT/ctypes_migration.sun`.

8. Analyze the system by issuing the following command:

   ```
   sdtbatch[.bat] -a systemPath/systemname.sdt -options
   optionFile.txt
   ```

   There are known restrictions that may make it necessary to manually make further changes of the case in some places in the system. Known problems are:

   – Text in class symbols are ignored.

   – Words spanning two symbols using the possibility to divide a word with '_' are not updated.

   – When generating SDL PR files from SDL GR files some words are trimmed for spaces. Those may not be handled correctly.

We recommend that you update the system further until you have no known errors in the system when running in case-sensitive mode before continuing the process to migrate to CPP2SDL.

Changed Tool Integration

CPP2SDL is integrated with the Organizer and the SDL Editor in a slightly different way than H2SDL is. Instead of placing the header files one by one at root level in the Organizer view, it is now possible to define a set of header files which will be translated by CPP2SDL as a group. Each such set is organized by an import specification which controls what declarations should be translated and how they should be translated. The SDL/PR file that results from the processing of an import specification is represented by means of a PR symbol in the SDL Editor.

1. Remove all header files from the root level in the Organizer view. Also remove the symbol for the ctypes package.
2. In the extended heading of the SDL system diagram, remove the use clauses of the packages, which were generated by H2SDL for the header files. Also remove the use clause of the ctypes package.

> **Note:**
>
> The ctypes package must not be used with CPP2SDL as it is designed exclusively for H2SDL. A set of SDL/PR files plays the same role as the ctypes package and will be included by selecting an option in an import specification as described in step 5.

Now all dependencies on H2SDL have been removed, and the specification may be set up for using CPP2SDL instead.

3. Add an import specification at system level in the Organizer view. Do this by placing a PR symbol in the SDL system diagram. Then double-click the PR symbol, and select it to be a C import specification in the dialog that appears.
4. Add the removed header files to the import specification, by selecting the import specification and using the Add Existing command in the Organizer. The symbols for the added header files will appear below the import specification symbol in the Organizer.
5. Go to the CPP2SDL Options dialog for the import specification and set the option[1] to generate SDL representations for fundamental types.

When the above steps have been performed, the SDL specification should be equivalent to the original specification from a tool integration

point-of-view. However, while doing these changes there are a few optimizations that are enabled by CPP2SDL which should be considered. We will look at these later in "Configuring CPP2SDL Translation" on page 880.

Differences in Translation Rules

Although the translation rules implemented by CPP2SDL in many ways are similar to the ones implemented by H2SDL, they are not identical. Therefore, it may be necessary to do some modifications in the SDL specification.

The following sections follow the presentation in chapter 14, *The CPP2SDL Tool*.

Names

The names of generated SDL identifiers may differ in prefixes and suffixes. All references to such identifiers from the SDL specification should be updated, if needed.

1. Remove suffixes generated by H2SDL to handle case-sensitivity. These suffixes have the format "_*i*", where *i* is a natural number.
2. Remove prefixes generated by H2SDL to handle combinations of underscores that previously were not supported. These prefixes have the format "zz_UScr_*ij*_", where *i* and *j* are natural numbers. Also change the name of these identifiers to be the same as the corresponding C identifiers.
3. Change prefixes generated by H2SDL to handle SDL keywords. These prefixes have the format "zz_CCod_" and should be replaced with "keyword_".

Hint:

An alternative to changing all keyword prefixes manually, is to set the keyword prefix to be "zz_CCod_" in the CPP2SDL Options dialog.

1. This option plays the same role as the ctypes package did for H2SDL. When it is set, the generated SDL will include suitable SDL/PR files with SDL representations of fundamental C/C++ types and type declarators (e.g. int, char, pointers and arrays). If set at more than one specification, errors will occur. Therefore, set this option at the broadest scope in which it will be used.

Fundamental Types

The ctypes package used by H2SDL, is replaced by one of two sets of SDL/PR files, depending on the used input language; BasicCTypes and CPointer, or BasicC++Types and C++Pointer. They are included automatically in the SDL generated by CPP2SDL if the *Generate SDL representations for fundamental types* is set. Refer to "Fundamental Types" on page 786 in chapter 14, *The CPP2SDL Tool* for more information.

The table below describes the differences between the H2SDL ctypes package and the corresponding CPP2SDL SDL/PR files. The most obvious difference is that the SDL names of the sorts that represent fundamental C/C++ types are more consistent and intuitive when CPP2SDL is used.

Also note that the `CharStar` and `VoidStarStar` sorts of ctypes (representing `char*` and `void**` in C/C++) are not predefined in the CPP2SDL included SDL/PR files. See "Predefined Pointer Types" on page 877 and "Predefined Operators" on page 877.

C/C++ Fundamental Type	SDL Sort when using H2SDL	SDL Sort when using CPP2SDL
`signed int` `int`	`Integer`	`int`
`unsigned int` `unsigned`	`UnsignedInt`	`unsigned_int`
`signed long int` `signed long` `long int` `long`	`LongInt`	`long_int`
`unsigned long int` `unsigned long`	`UnsignedLongInt`	`unsigned_long_int`
`signed short int` `signed short` `short int` `short`	`ShortInt`	`short_int`
`unsigned short int` `unsigned short`	`UnsignedShortInt`	`unsigned_short_int`

C/C++ Fundamental Type	SDL Sort when using H2SDL	SDL Sort when using CPP2SDL
signed long long int signed long long long long int long long	LongLongInt	long_long_int
unsigned long long int unsigned long long	UnsignedLongLongInt	unsigned_long_long_int
char	Character	char
signed char	Character	signed_char
unsigned char	Octet	unsigned_char
char *	CharStar	N/A
float	Float	float
double long double	Real	double
bool	N/A	bool
wchar_t	N/A	wchar_t
void *	VoidStar	ptr_void
void **	VoidStarStar	N/A

4. Update all references to sorts corresponding to fundamental types, with the names of the corresponding SDL sorts generated by CPP2SDL.

Hint:

An alternative to changing all these references manually is to define a set of syntypes mapping the old names to the new ones.

Type Declarators

The names of SDL sorts representing pointer and array types have been changed to be more intuitive and shorter when CPP2SDL is used.

5. Update all references to sorts corresponding to pointer types by changing the old pointer prefix “Ref_*filename*_” to “ptr_”.

6. Update all references to sorts corresponding to array types by changing the old array prefix “CA_*filename_i*_X_” to “arr_*i*_”, where *i* is the number of elements in the array.

As the H2SDL array prefix “CA_*filename_i*_X_” would result in very long SDL sort names when translating multi-dimensional arrays, H2SDL generates syntypes to shorten the sort names. These syntypes are named “MA_*filename_j*”, where *j* is a natural number. Example 144 gives an example of this technique and the corresponding CPP2SDL translation.

Example 144: H2SDL translation of multi-dimensional array compared to CPP2SDL translation ––––––––––––––––––––––––––––––––––––

C header:

```
int func(int[2][3][4]);
```

H2SDL translation:

```
newtype CA_filename_4_X_Integer /*#SYNT*/
  CArray(4,Integer);
endnewtype CA_filename_4_X_Integer;

newtype CA_filename_3_X_CA_filename_4_X_Integer
/*#SYNT*/
  CArray(3,CA_filename_4_X_Integer);
endnewtype CA_filename_3_X_CA_filename_4_X_Integer;

syntype MA_filename_0 /*#SYNT*/ =
 CA_filename_3_X_CA_filename_4_X_Integer
endsyntype;

newtype CA_filename_2_X_MA_filename_0 /*#SYNT*/
  CArray(2,MA_filename_0);
endnewtype CA_filename_2_X_MA_filename_0;

procedure func;
  fpar
    fpar_0 CA_filename_2_X_MA_filename_0;
  returns Integer;
external;
```

CPP2SDL translation:

```
NEWTYPE global_namespace_ImpSpec /*#NOTYPE*/
  OPERATORS
    func : arr_2_arr_3_arr_4_int -> int;
/*#ADT(A(S) E(S) K(H))*/
ENDNEWTYPE global_namespace_ImpSpec;EXTERNAL 'C++';
NEWTYPE arr_4_int CArray( 4, int);
/*#ADT(A(S) E(S) K(H))*/
```

```
ENDNEWTYPE arr_4_int;EXTERNAL 'C++';
NEWTYPE arr_3_arr_4_int CArray( 3, arr_4_int);
/*#ADT(A(S) E(S) K(H))*/
ENDNEWTYPE arr_3_arr_4_int;EXTERNAL 'C++';
NEWTYPE arr_2_arr_3_arr_4_int CArray( 2,
arr_3_arr_4_int);
/*#ADT(A(S) E(S) K(H))*/
ENDNEWTYPE arr_2_arr_3_arr_4_int;EXTERNAL 'C++';
```

7. If syntypes for multi-dimensional arrays are present ("MA_*filename_j*"), replace them with one "arr_*i*_" prefix for each array dimension.

Predefined Pointer Types

The H2SDL ctypes package includes the definition of the SDL sorts `CharStar` and `VoidStarStar` corresponding to the C/C++ pointer types `char*` and `void**`. These predefined sorts are not supported by CPP2SDL. Instead CPP2SDL will translate these pointer types using the `Ref` generator as they are needed.

8. Update all references to the obsolete sort `CharStar` to the translated sort `ptr_char`.

9. Update all references to the obsolete sort `VoidStarStar` to the translated sort `ptr_ptr_void`.

Predefined Operators

The ctypes package also defines a number of obsolete conversion operators for the `Ref` generator and the `CharStar` and `VoidStarStar` sorts, these are replaced with the `cast` operator according to the table below.

H2SDL operator	CPP2SDL operator	Comment
`ref2vstar`	`cast`	
`vstar2ref`	`cast`	
`ref2vstarstar`	`N/A`	Use `cast(cast( ))` instead, i.e `<type>*` to `void*` to `void**`
`cstar2cstring`	`cast`	Defined in CharConvert.pr
`cstring2cstar`	`cast`	Defined in CharConvert.pr

H2SDL operator	CPP2SDL operator	Comment
cstar2vstar	cast	
vstar2cstar	cast	
cstar2vstarstar	N/A	Use cast(cast( )) instead, i.e <type>* to void* to void**
vstarstar2vstar	cast	
vstar2vstarstar	cast	

10. If the cstar2cstring or cstring2cstar operators are used, add a PR symbol and connect it to the file <installation directory>/include/ADT/CharConvert.pr.

11. Replace all occurrences of the obsolete conversions operators with the cast operator according to the table above.

Enumerated Types

H2SDL has two alternative translations of enum declarations; using integer synonyms or using newtype literals. CPP2SDL only supports the latter of these alternatives.

If H2SDL was configured to translate enum literals to integer synonyms instead of newtype literals, and such a generated SDL synonym is used as an arithmetic expression, then the EnumToInt operator generated by CPP2SDL should be invoked on the newtype literal that corresponds to that synonym. See "Enumerated Types" on page 790 in chapter 14, *The CPP2SDL Tool* for more information.

Another difference in the translation of enumerated types is that H2SDL adds an Enum_ prefix to the generated newtype, while CPP2SDL uses the name as it is.

12. Remove all Enum_ prefixes from identifiers in references to newtypes representing enumerated types.

Functions

While H2SDL translates function prototypes to SDL procedures, CPP2SDL translates them to operators.

13. Convert all calls to procedures that represent C functions into operator calls by removing the `call` keyword.

14. If the Procedure call symbol is used to call procedures representing C functions, replace the Procedure call symbol with a Task symbol.

Note:

A procedure without parameters is called differently than an operator without parameters:

```
/* Calling a procedure p without parameters. */
task call p();
/* Calling an operator o without parameters. */
task o;
```

Variables

H2SDL does not translate C variables directly, but generates two procedures for getting and setting the value of the variable. CPP2SDL, on the other hand, generates external SDL variables, which is a tool-specific SDL extension.

15. For each accessed variable `v`, replace all procedure calls to `Set_v(`*`value`*`)` with the assignment `v := `*`value`*. Replace procedure calls to `Get_v()` with the variable name `v`.

Note:

CPP2SDL will only translate variables when the Import Specification is placed in a SDL diagram that allows declaration of variables.

Structs and Unions

With CPP2SDL, the names of newtypes generated from structs and unions are retained, while H2SDL adds a `Struct_` or a `Union_` prefix to these names.

16. Remove all `Struct_` and `Union_` prefixes from identifiers in references to newtypes representing structs or unions.

Dynamic Memory Management

The definition of the Ref generator that was used with H2SDL only contained an operator (`Make!`) or literal (`Alloc`) for instance allocation,

while instance deallocation was made with a `Free` procedure, defined outside the generator. With CPP2SDL, the `Free` procedure has been replaced with a `free` operator in the Ref generator.

17. Convert all calls to the `Free` procedure into calls to the `free` operator by removing the **call** keyword and changing `Free` to `free`.

Configuring CPP2SDL Translation

While H2SDL performs a full translation of all supported C constructs to the system scope of the SDL specification, the Import Specification makes it possible to define what CPP2SDL translates and where to import it.

Additionally, it is also possible to use multiple Import Specifications to import C declarations to different scopes in the SDL specification.

> **Note:**
>
> The SDL language does not contain any construct like the C `#ifdef` construct, so take care not to import any declarations more than once if you are planning to use multiple Import Specifications.

Where to place the Import Specifications

For example, instead of always placing an import specification at system level, it should be placed at the most narrow scope that encloses all SDL usages of the declarations in the header files that are present under that import specification.

CPP2SDL Options

H2SDL used a centralized setting controlling which preprocessor and what preprocessor options to use for preprocessing input headers. CPP2SDL is more flexible, and makes it possible to specify these settings for each import specification.

1. Open the CPP2SDL Options dialog for each import specification, and enter the preprocessor and preprocessor options to use for preprocessing the headers grouped by that import specification.

Also note that H2SDL defines the macro `__H2SDL__` while preprocessing the C header files. When using CPP2SDL this macro is replaced by the `__CPP2SDL__` macro.

2. If the __H2SDL__ macro is used in your C headers change this to __CPP2SDL__.

> **Hint:**
>
> An alternative to changing the __H2SDL__ macro is to use the pre-processor options in the CPP2SDL Options dialog to define the __H2SDL__ macro.

For more information on the options available in the CPP2SDL Options dialog please refer to "Setting CPP2SDL Options in the Organizer" on page 766 in chapter 14, *The CPP2SDL Tool*.

What to Translate

The TRANSLATE section of the Import Specification controls which C identifier that should be translated, by default CPP2SDL will also translate any declarations that these identifiers are depending on.

With the TRANSLATE section it is also possible to add type declarators to types and supply prototypes for ellipsis functions making these available in SDL.

Read more about the TRANSLATE section in "Import Specifications" on page 776 in chapter 14, *The CPP2SDL Tool*.

New Build Options

Contrary to H2SDL, CPP2SDL requires the generated SDL to be analyzed as case sensitive SDL.

3. In the Analyzer Options dialog, set the Case sensitive SDL option.

It should now be possible to analyze the SDL specification.

When the SDL specification has been accepted by the Analyzer, it should also be possible to generate code without further modification. Note, however, that using the Targeting Expert is the recommended way of generating code when CPP2SDL is used. This means that if a makefile or makefile template was used with H2SDL to link the external object files to the generated application, this could now be achieved in an easier way using the Targeting Expert. For more information, see "Generated Makefile" on page 2997 in chapter 59, *The Targeting Expert*.

Chapter 15 CPP2SDL Migration Guide

Chapter

16 *CIF Converter Tools*

This chapter is a reference manual for the CIF2SDT and SDT2CIF converter tools. The CIF2SDT tool converts CIF (the Common Interchange Format as defined in the Z.106 recommendation) files to binary format files, while SDT2CIF tool converts binary files or system files to CIF files.

CIF is a human readable text format for storing SDL diagrams, making it possible to store SDL diagrams in version control systems that only accept text files. In addition, it provides a base for converting diagrams to and from other SDL toolsets. The CIF2SDT and SDT2CIF tools bring the advantage of CIF to the SDL Suite tools.

Introduction

Common Interchange Format

CIF (Common Interchange Format) is a text format specified by ITU-T (International Telecommunication Union) in the recommendation Z.106 to ease the interchange of graphical SDL specifications. SDL-CIF is an extension to SDL/PR and is based on the SDL/PR syntax and can be read and written by tools as well as users. The CIF constructs are expressed as comments preceding the PR code. The advantages of such a text format are:

- SDL diagrams may be stored in version control systems that only accept text files.
- The format is human readable.
- It makes it possible to convert diagrams, including their layout, to and from other SDL toolsets that support CIF.
- It also makes it possible to revert to previous versions of the SDL Suite, by using CIF as an intermediate format and re-opening the CIF files in an older version of the SDL Suite to become SDL-GR.

However, CIF cannot be used as a storage format for files in the SDL Suite as it is required that specification stored in CIF file must be complete and correct.

CIF <-> SDT Converters

The CIF2SDT and SDT2CIF converters make the advantages of the CIF format available to the SDL Suite tools. The converters are implemented as a binary that can be run in a textual, command-line mode from the OS prompt, or as separate applications with graphical user interfaces **(in Windows)**. The converter tools can also be launched from the Organizer using the menu choices *Convert GR to CIF* and *Convert CIF to GR* from the *Generate* menu.

The conversion between CIF and the binary format is performed as follows:

- When converting to CIF, the tool SDT2CIF is used. CIF comments will be generated close to the corresponding SDL/PR code.

- When converting from CIF, the tool CIF2SDT is used. Both CIF comments and SDL/PR code are considered. CIF comments control conversion, while the text for symbols is extracted from the SDL/PR code.

Note:

The CIF2SDT or SDT2CIF tools require the SDL Suite environment when running; meaning that there should be enough licenses available to be able to convert to/from CIF.

On UNIX, when running the CIF converters without having started the SDL Suite tools, the environment variable $`telelogic` must be set. This will be set by souring the file `telelogic.sou` in the installation directory.

Generally, one CIF file contains several SDL diagrams. However, the placement of generated diagrams can be specified by the user.

CIF2SDT Converter Tool

The CIF2SDT tool converts CIF files to binary format files. It can convert one or more specified CIF files or all CIF files in a specified directory.

The CIF2SDT converter never overwrites existing data files. If the converter finds that the file it was going to write already exists, it generates a new name for the output file by replacing the two last characters at the end of the file name.

The CIF2SDT converter can be started in the following ways:

- From the OS command line; see "Command Line Syntax" on page 886.
- As a separate application with a graphical user interface **(Windows only)**; see "Graphical User Interface (Windows only)" on page 890.
- From the Organizer's *Generate* menu choice *Convert CIF to GR*.
  - **On UNIX**, a dialog is then opened; see "Convert CIF to GR Dialog (UNIX only)" on page 889.
  - **In Windows**, the separate application is then started; see "Graphical User Interface (Windows only)" on page 890.

Command Line Syntax

The CIF2SDT tool can be invoked from the OS prompt as follows:

```
cif2sdt [-v[ersion]] [-h] [-r] [-k]
[ -o <output file name> ]
( <CIF file> | <directory> )*
```

The meaning of the command-line options is given in the following sections.

(Windows only) When started without options or with -r option only, the Graphical User interface will be started, otherwise the application will execute as a command line application. However, to be able to get the output in the command window when run as a command line application you must use the special application conspawn as a wrapper (assuming `C:\IBM\Rational\SDL_TTCN_Suite6.3` is the installation directory)

```
C:\IBM\Rational\SDL_TTCN_Suite6.3\bin\wini386\conspawn.exe -b

C:\IBM\Rational\SDL_TTCN_Suite6.3\bin\wini386\cif2sdt.exe
myciffile
```

When using the converter as a command line application and invoking it within emacs path names in input files are lost. Therefore be sure to do a “cd” to the directory where the files are before invoking the converter.

Command Line Options

- `-v` or `-version` (show the version)

  This option displays the version number of the tool.

- `-h` (show the command line syntax)

  This option displays a help message about the command line syntax.

- `-r` (reuse a started session)

  This option allows to reuse and connect to an already started session if it exists. The default behavior is to start a new session.

- `-k` (keep original file name)

  If the CIF file being converted was converted from an SDL diagram, the converter will use the original file name for the output file.

Note:

This option will only work if the CIF file contains the CIF comment OriginalFileName.

- `-o` (save the generated files under the name specified)

  When given this option, the SDT2CIF converter will save the generated diagrams in the file named `<output file name>`.

Examples of Usage

Example 145 ————————————————————————————

```
cif2sdt myfile.cif
```

The CIF2SDT tool will convert the diagrams contained in the CIF file `myfile.cif` to the file `myfile.sif` (if a file named `myfile.sif` does

not already exist), `myfile01.sif` (if a file named `myfile.sif` already exists, but the file `myfile01.sif` does not), `myfile02.sif` and so on.

Example 146 ----------

```
cif2sdt -o lift.sbk lift.cif
```

The CIF2SDT tool will convert the diagrams contained in the CIF file `lift.cif` to binary format and put it into an SDL block file named `lift.sbk` (again, if the file named `lift.sbk` does not already exist – otherwise, the name will be modified and a corresponding message issued).

Example 147 ----------

```
cif2sdt mycifdir
```

(where `mycifdir` specifies a directory name)

The CIF2SDT tool will search for CIF files in the directory named `mycifdir`, convert all CIF files found to binary format files.and put the converted diagrams in the directory `mycifdir`. For information on how the search for CIF files is done, see "How the Converter Works" on page 892.

Example 148 ----------

```
cif2sdt myblock.cbk myprocss.cpr myservce.csv
```

The CIF2SDT tool will convert the CIF files `myblock.cbk`, `myprocss.cpr` and `myservce.csv` and put the converted diagrams into the binary files `myblock.sbk`, `myprocss.spr` and `myservce.ssv`.

Example 149 ----------

```
cif2sdt mysystem.csy mycifdir
```

(where `mycifdir` specifies a directory name)

The CIF2SDT tool will convert the CIF file `mysystem.csy` and put the converted diagrams into an system file, then it will search the directory `mycifdir` and convert all CIF files found in this directory.

Convert CIF to GR Dialog (UNIX only)

On UNIX, the CIF2SDT converter can also be started by selecting *Convert CIF to GR* from the Organizer's *Generate* menu. This opens the *Convert CIF to GR* dialog:

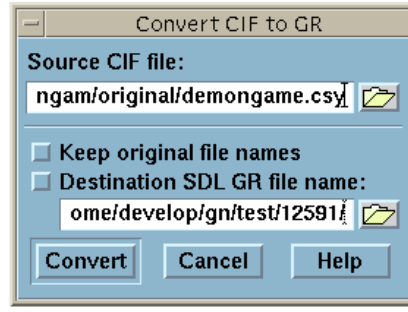


Figure 169: The Convert CIF to GR dialog

The dialog functions as a front-end to the `cif2sdt` converter described in "Command Line Syntax" on page 886. The only functionality **not** supported by the dialog is the possibility to specify a directory name as input to the conversion.

- *Source CIF file*

  This text field specifies the name of the CIF file to be converted (a directory name **cannot** be specified). To select a file using a standard file selection dialog box, press the folder button located to the right of the text field.

- *Keep original file names*

  If this option is enabled, the converter will preserve the original file name (i.e. the file from which this CIF file was generated).

> **Note:**
> This option will only work if the CIF file contains the CIF comment OriginalFileName.

- *Destination SDL/GR file name*

  This text field specifies the name of the file which will contain the converted diagrams. To select a file using a standard file selection

dialog box, press the folder button located to the right of the text field.

- *Convert*

  Clicking this button will initiate the conversion, using the options currently displayed in the dialog. Messages issued by the converter will appear in the Organizer Log window. The possible messages are listed in "Messages from CIF2SDT Converter" on page 893.

Graphical User Interface (Windows only)

In Windows, the CIF2SDT converter is either started outside the SDL Suite by using the `cif2sdt` executable in the release, or by selecting *Convert CIF to GR* from the Organizer's *Generate* menu. It provides a graphical user interface to control the conversions:

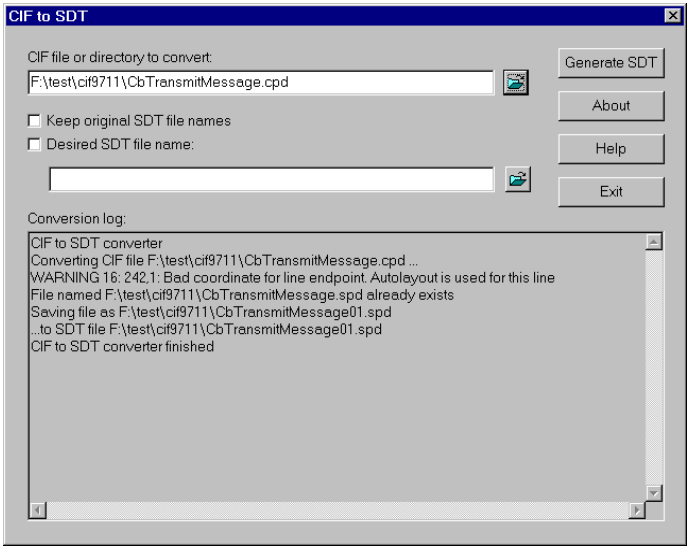


Figure 170: CIF2SDT Graphical User Interface

To convert a CIF file or all CIF files in a directory, perform the following steps:

1. Type a CIF file or directory name in the text field under *CIF file or directory to convert*.
   - To select a file using a standard dialog box, click the folder button which is located to the right of the text field.
2. Specify the desired options by clicking on check boxes which specify conversion options; look for option explanations in "Converter Options" on page 891.
3. Click the *Generate SDT* button.

After these steps, the conversion will proceed, possibly printing warning/errors/information messages into the *Conversion log* text box.

After the conversion has been performed, the CIF2SDT converter can be used again to convert more CIF files or directories.

Converter Options

- *CIF file or directory to convert*

  This text field specifies the name of the CIF file or directory to be converted. To select a file using a standard file selection dialog box, press the folder button located to the right of the text field.

- *Keep original SDT file names*

  If this option is enabled, the converter will preserve the original file name (i.e. the file from which this CIF file was generated).

> **Note:**
> This option will only work if the CIF file contains the CIF comment OriginalFileName.

- *Desired SDT file name*

  This text field specifies the name of the file which will contain the converted diagrams. To select a file using a standard file selection dialog box, press the folder button located to the right of the text field.

- *Conversion log*

This text box contains warning/error/information messages issued by the converter during the last conversion. The possible messages are listed in "Messages from CIF2SDT Converter" on page 893.

- *Generate SDT*

  Clicking this button will initiate the conversion, using the options currently displayed in the window.

- *Exit*

  Clicking this button exits the converter.

How the Converter Works

First of all, the converter analyzes the input and builds a list of files to convert. If one or a set of files was specified, it adds all the files specified to the list of files to convert. If a directory was specified, the converter searches in the specified directory for files with the extensions `.cif, .cun, .csy, .cbk, .csu, .cpr, .csv, .cpd, .cmc, .cst, .cbt, .cpt, .cvt, .cop` (which are assumed to contain CIF diagrams) and adds the files found to the list of files to convert. The extension for the output file is formed by replacing the first character of the extension of the input file with the character 's'.

When the list of files to convert is ready, the converter tries to convert each file in the list. The file being converted is parsed, analyzed, transformed, and saved. During parsing, CIF comments take precedence over the PR text, which is supposed to contain additional attributes for CIF objects. If the converter finds an error in the PR text, it proceeds by printing a warning message about that fact, using only the information extracted from the CIF comments.

The advantage of the CIF2SDT converter is that it supports incomplete PR text. For example, it can handle empty text of symbols, empty signal lists, and empty gate constraints. However, there is an exception when the PR text is required to be correct: after the "Diagram Start" CIF comment.

If a line is found that does not have the correct breakpoints (to be correct, it should be a point on the symbol border), an auto layout method is used to place the line.

The converter never overwrites any existing files. If the file to be written gets the same name as an existing file, a new name is generated for the output file to avoid overwriting existing files.

Messages from CIF2SDT Converter

The CIF2SDT converter issues an *information message* when it needs to inform the user about something; for example, when it changes the output file name in order not to overwrite an existing file. It also issues a *warning message* when some non-fatal error is found; for example, when some symbol on a diagram occurs in a wrong context. It prints an *error message* when it is not possible to continue the conversion; for example, when the input file is wrong or corrupt.

Message Format

The general format for warning/error messages is the following:

```
ERROR <error code>: [<line>, <column>:] <error text>
<additional information>
WARNING <warning code>: [<line>, <column>:] <warning
text> <additional information>
```

The `<error code>` specifies the error code which can be used to find the warning/error explanation (see the following sections). The `<line>` and `<column>` specify a position in the source CIF file where the error has occurred. The `<error text>` gives a short explanation of what is wrong. The `<additional information>` specifies additional information about the error (for example, it can specify the name of an endpoint constraint that could not be bound).

The list of possible warning/error messages follows in the next sections.

List of Error Messages

`Error 1: Arguments required` (UNIX only)

This error indicates that no command-line arguments were given to the CIF2SDT converter, which thus cannot continue processing. To remedy the situation, specify one or more file/directory name(s) to convert.

`Error 2: Illegal option` **(UNIX only)**
This message is issued when an invalid option is found in the command line. To remedy the situation, supply an appropriate option instead of the invalid one.

`Error 3: Duplicate option` **(UNIX only)**
This message is issued when a duplicate command-line option is found. The command-line options can be specified only once. To remedy the situation, remove duplicated options from the command line.

`Error 4: Missing output file` **(UNIX only)**
This message is issued when the option `-o` has been specified, but no output file name follows. To remedy the situation, supply an output file name after the `-o` option.

`Error 5: Illegal command line syntax` **(UNIX only)**
This message is issued when the command line is found not to obey the command-line syntax. To remedy the situation, make the command line conform to the command-line syntax.

`Error 6: Impossible to connect to SDT PostMaster`
This message is issued when the CIF2SDT converter cannot connect to the PostMaster.

In Windows, the CIF2SDT converter requires the PostMaster to be running in order to perform conversion. Start the Organizer and try again.

On UNIX, the most likely cause of this message is that either the path to the SDL Suite tools is not in the search path, or the maximum number of licenses is reached. To remedy the situation, ensure that the SDL Editor can be started (i.e. it is in the search path) and that there are enough licenses available.

`Error 7: Error creating diagram`
This message is issued when the CIF2SDT converter cannot create a diagram in the SDL Editor. This message is most probably caused by a corrupt input CIF file. To remedy the situation, correct the CIF file.

Error 8: Error creating page

This message is issued when the CIF2SDT converter cannot create a page in the SDL Editor. This message is most probably caused by a corrupt input CIF file. To remedy the situation, correct the CIF file.

Error 9: Error creating symbol

This message is issued when the CIF2SDT converter cannot create a symbol in the SDL Editor. This message is most probably caused by a corrupt input CIF file. To remedy the situation, correct the CIF file.

Error 10: Error saving diagram in file

This message is issued when the CIF2SDT converter cannot save the resulting diagram on the output file. This message may be caused by an invalid output path/file name. If this is the case, specify a valid path. This message may also be caused by insufficient disk space available. If this is the case, free some disk space.

Error 11: Size of symbol is not specified

This message is issued when the CIF2SDT converter encounters a symbol without an explicit size specified and without any default size specified in a diagram in the input CIF file. This is illegal according to the Z.106 standard. To remedy the situation, correct the source CIF file.

Error 12: Cannot bind gate reference

This message is issued when the CIF2SDT converter cannot bind a gate reference to a list of connections. This means the input file is corrupt. To remedy the situation, correct the source CIF file.

Error 13: Cannot bind connect for...

This message is issued when the CIF2SDT converter cannot bind a connection statement with a list of channels/signal routes. This means the input file is corrupt or illegal. To remedy the situation, correct the source CIF file.

Error 14: Cannot bind FROM endpoint for...

This message is issued when the CIF2SDT converter cannot bind a FROM endpoint of a channel or signal route to a block or process. This means that the input CIF file is corrupt or illegal. To remedy the situation, correct the CIF file.

Error 15: Cannot bind TO endpoint for...

This message is issued when the CIF2SDT converter cannot bind a TO endpoint of a channel or signal route to a block or process. This means that input CIF file is corrupt or illegal. To remedy the situation, correct the CIF file.

Error 16: Cannot bind endpoint of line

This message is issued when the CIF2SDT converter encounters a flow line statement and cannot find symbols which are supposed to be connected. To remedy the situation, correct the CIF file.

Error 17: Dashed should be used if keyword Adding is used for gate

Error 18: Wrong first endpoint in gate

Error 19: Gate constraint symbol is omitted for gate

These three messages mean that the source CIF file contains contradictory CIF comments and SDL/PR and is thus corrupt. To remedy the situation, correct the CIF file.

Error 20: Wrong page name

This message is issued when the CIF2SDT converter encounters a CIF PageSwitch comment in a source file that references an undefined page in the diagram. To remedy the situation, correct the CIF file.

Error 21: Wrong number of points in pointlist for gate

This message is issued when the CIF2SDT converter encounters a CIF Gate comment with a pointlist consisting of more than two points. This is a violation of the Z.106 standard, the input file is thus corrupt. To remedy the situation, correct the CIF file.

Error 22: Create line can occur only inside block (type) without decomposition

This message means that the source CIF file is corrupt. To remedy the situation, correct the source CIF file.

Error 23: Flowline can occur only inside diagrams with process body

This message means that the source CIF file is corrupt. To remedy the situation, correct the source CIF file.

Error 24: Symbol cannot occur in this context

This message means that the CIF2SDT converter encountered a graphical symbol in a wrong context in the source diagram (for example, a start symbol in a system diagram). To remedy the situation, correct the source CIF file.

Error 25: Syntax error

This message means that the CIF2SDT converter encountered a violation of the CIF syntax in the source CIF file. To remedy the situation, correct the source CIF file.

Error 26: Illegal name of output file

This message means that either no unique name was found for the output file or there was an input/output error during saving of the output file. To remedy the situation, try to convert the diagrams to a file with a different name (**on UNIX**, see the `-o` option).

Error 27: Cannot bind connection point text position

This message means that the position of text in a connection statement could not be bound. To remedy the situation, try to move the text of connection symbol in the source diagram, then re-run the converter.

Error 28: Wrong page type(s) on the diagram

This message means that the CIF2SDT converter found a process diagram with several pages of different types, which is a violation of the Z.106 standard. To remedy the situation, correct the CIF file.

Error 29: Analysis of CIF failed

This message means that the CIF2SDT converter failed to analyze the input CIF file. In this case nothing is generated in the output file. To remedy the situation, correct the CIF file.

Error 30: Wrong syntax of extended task

This message means that the CIF2SDT converter found a syntax error in the PR text of an extended task symbol. The incorrect text is printed as the rest of the error message. Either the left or right curly bracket is missing or they appear in the wrong place in the PR text. To remedy the situation, correct the CIF file.

List of Warning Messages

Warning 2: No input files

This warning is issued when the CIF2SDT converter finds out that no valid file or directory names were specified on the command line. To remedy the situation, ensure that the specified path/file names are correct.

Warning 3: Cannot convert file or directory

This warning is issued when the CIF2SDT converter cannot determine if the specified file is a CIF file or a directory. This is most probably caused by an invalid file/path name. To remedy the situation, ensure that the specified path/file names are correct.

Warning 4: SDLE says: ...

This warning is issued when the CIF2SDT converter receives a reply from the SDL Editor with a message explaining the reason of the error. To see the more detailed explanation of the reason for the error, see the text following the colon.

Warning 5: Empty diagram

This warning means that no diagrams are contained in the source CIF file.

Warning 6: Sorry, Select symbol is not supported in SDT SDLE

Since Select symbol is not supported in the SDL Editor, the converter has no option but to ignore it.

Warning 7: ... cannot contain ... text position specification, specification ignored

Warning 7: ... text placement specification is already specified, ignoring extra specification

Warning 8: This is location of previous specification

These warning messages mean that the source CIF file is corrupt. To remedy the situation, correct the CIF file.

Warning 9: Parsing of PR has failed

This means that there is incorrect SDL/PR after some CIF comment in the source CIF file. To remedy the situation, correct the CIF file.

Warning 10: Dashed attribute synthesized

Warning 11: Dashed attribute ignored

Warning 12: Ignoring gate references

These three messages mean that there are errors in source CIF file that have been automatically corrected.

Warning 13: Integer value expected, zero assumed

This message means that some other text was encountered instead of integer number in the source CIF file, thus the value has been assumed to be zero.

Warning 14: PR is not allowed in ...

This message means that the CIF2SDT converter encountered SDL/PR where it is not allowed to be. This also indicates that the source CIF file is corrupt. To remedy the situation, correct the CIF file.

Warning 15: Descriptors are not allowed (in typebased system)

This means that the source CIF file contained a type based system diagram with some symbols in it, which is a violation of the Z.106 standard. These symbols have been ignored.

```
Warning 16: Bad coordinate for line endpoint.
Autolayout is used for this line
```

This means that a coordinate for a line endpoint has been found but this coordinate does not fall on the perimeter of the symbol that is connected to the line. Autolayout of the line will be used to place the line.

SDT2CIF Converter Tool

The SDT2CIF tool converts binary format files to CIF files. It can convert one or more specified binary format files, SDL system files or all SDL diagram files in a specified directory.

The SDT2CIF converter also supports the mixed platform feature of the SDL Suite (for more information on this topic, see "Windows and UNIX File Compatibility" on page 215 in chapter 2, *The Organizer*. If the converter finds a `[DRIVES]` section during the conversion of an SDL system file, it assumes that this system file is used on mixed platforms. This makes the converter change all characters in output file name to lower case **(on UNIX)** or replace the part of the path by the drive letter **(in Windows)**.

The SDT2CIF converter never overwrites existing data files. If the converter finds that the file it was going to write already exists, it generates a new name for the output file by replacing the two last characters at the end of the file name.

The SDT2CIF converter can be started in the following ways:

- From the OS command line; see "Command Line Syntax" on page 901.
- As a separate application with a graphical user interface **(Windows only)**; see "Graphical User Interface (Windows only)" on page 906.
- From the Organizer's *Generate* menu choice *Convert GR to CIF*.
  - **On UNIX**, a dialog is then opened; see "Convert GR to CIF Dialog (UNIX only)" on page 904.
  - **In Windows**, the separate application is then started; see "Graphical User Interface (Windows only)" on page 906.

Command Line Syntax

On UNIX, the SDT2CIF tool can be invoked from the OS prompt as follows:

```
sdt2cif [-v[ersion]] [-h] [-r] [-g] [-i] [-s]
[-o <output file name>] ( <binary diagram file> |
<system file> | <directory> )*
```

(Windows only) When started without options or with -r option only, the Graphical User interface will be started, otherwise the application will execute as a command line application. However, to be able to get the output in the command window when run as a command line application you must use the special application conspawn as a wrapper (assuming `C:\IBM\Rational\SDL_TTCN_Suite6.3` is the installation directory)

```
C:\IBM\Rational\SDL_TTCN_Suite6.3\bin\wini386\conspawn.exe -b

C:\IBM\Rational\SDL_TTCN_Suite6.3\bin\wini386\cif2sdt.exe
myciffile
```

When using the converter as a command line application and invoking it within emacs path names in input files are lost. Therefore be sure to do a “cd” to the directory where the files are before invoking the converter.

The meaning of the command-line options is given in the following sections.

Command Line Options

- `-v` or `-version` (show the version)

  This option displays the version number of the tool.

- `-h` (show the command line syntax)

  This option displays a help message about the command line syntax.

- `-r` (reuse a started session)

  This option allows to reuse and connect to an already started session if it exists. The default behavior is to start a new session.

- `-g` (include graphical SDT references)

  This option directs the SDT2CIF converter to include graphical SDT references into the generated CIF file. If `-g` is given, graphical references are stored as SDL/PR comments; otherwise they are omitted.

- `-i` (omit CIF comments)

  When using this option, only SDL/PR will be generated by SDT2CIF.

- `-s` (single file)

  When using this option, all diagrams will be saved on one single file. The name of the file is the same name as the first CIF file would get after conversion if this option was not used.

- `-o` (save the generated files under the name specified)

  When given this option, the SDT2CIF converter will save the generated diagrams in the file named `<output file name>`.

Examples of Usage

Example 150 ––

```
sdt2cif mysystem.sdt
```

where `mysystem.sdt` has the following diagrams:

```
myblock.sbk
myprocss.spr
myservce.ssv
myproced.spd
```

The SDT2CIF tool will convert all the diagrams referred to in the system file and save them in the CIF format under transformed names:

```
myblock.sbk    -> myblock.cbk
myprocss.spr   -> myprocss.cpr
myservce.ssv   -> myservce.csv
myproced.spd   -> myproced.cpd
```

Example 151 ––

```
sdt2cif mysdtdir
```

(where `mysdtdir` specifies a directory name)

The SDT2CIF tool will look in the directory `mysdtdir` and convert all binary files (but not system files – this would cause double conversion) to the corresponding CIF files.

Example 152 ––

```
sdt2cif myblock.sbk myprocss.spr myservce.ssv
```

The SDT2CIF tool will convert the three binary files to the CIF files `myblock.cbk`, `myprocss.cpr` and `myservce.csv`.

Example 153 ----------

```
sdt2cif -o blocks.cif block_a.sbk block_b.sbk
block_c.sbk
```

The SDT2CIF tool will convert the three binary files `block_a.sbk`, `block_b.sbk`, `block_c.sbk` and put all diagrams converted into the file `blocks.cif`.

Example 154 ----------

```
sdt2cif -s block_a.sbk block_b.sbk block_c.sbk
```

The SDT2CIF tool will convert the three binary files `block_a.sbk`, `block_b.sbk`, `block_c.sbk` and put all diagrams converted into the file named `blocks_a.cbk` (resulting from the name under which the first file in the list of files to convert would be saved if the `-s` switch was not specified).

Convert GR to CIF Dialog (UNIX only)

On UNIX, the SDT2CIF converter can also be started by selecting *Convert GR to CIF* from the Organizer's *Generate* menu. This opens the *Convert GR to CIF* dialog:

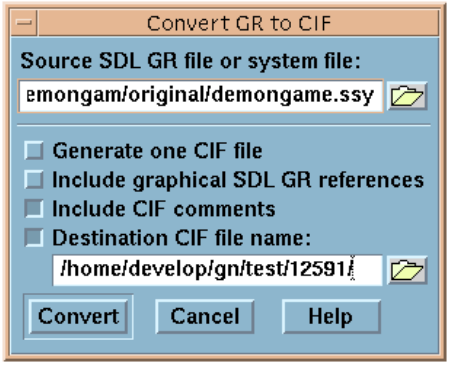


Figure 171: The Convert GR to CIF dialog

The dialog functions as a front-end to the `sdt2cif` converter described in "Command Line Syntax" on page 901. The only functionality **not**

supported by the dialog is the possibility to specify a directory name as input to the conversion.

- *Source SDL/GR file or system file*

  This text field specifies the name of the SDL/GR file or system file to be converted (a directory name can **not** be specified). To select a file using a standard file selection dialog box, press the folder button located to the right of the text field.

- *Generate one CIF file*

  If this option is enabled, the converter will save all converted diagrams into one file. The name of this file is determined from the *Destination CIF file name* check box state. If the check box is not selected, the name of the file will be the same as the name the first file being converted would get. This option is disabled by default. If the check box is selected, the name of the file with diagrams will be the name specified in the *Destination CIF file name* text field.

- *Include graphical SDL/GR references*

  This option directs the SDT2CIF converter to include graphical SDT references in the generated CIF files (see chapter 18, *SDT References* for more information on this topic). If this option is enabled, graphical SDT references are stored as SDL/PR comments; otherwise they are omitted. This option is disabled by default.

- *Include CIF comments*

  If this option is disabled, only SDL/PR will be generated to output file; otherwise CIF comments will be included. This option is enabled by default.

- *Destination CIF file name*

  This text field specifies the name of the file that will contain the converted diagrams. To select a file using a standard file selection dialog box, press the folder button located to the right of the text field.

- *Convert*

  Clicking this button will initiate the conversion, using the options currently displayed in the dialog. Messages issued by the converter will appear in the Organizer Log window. The possible messages are listed in "Messages from SDT2CIF Converter" on page 908.

Graphical User Interface (Windows only)

In Windows, the SDT2CIF converter is either started outside the SDL Suite by using the `sdt2cif` executable in the release, or by selecting *Convert GR to CIF* from the Organizer's *Generate* menu. It provides a Graphical User Interface to control the conversions:

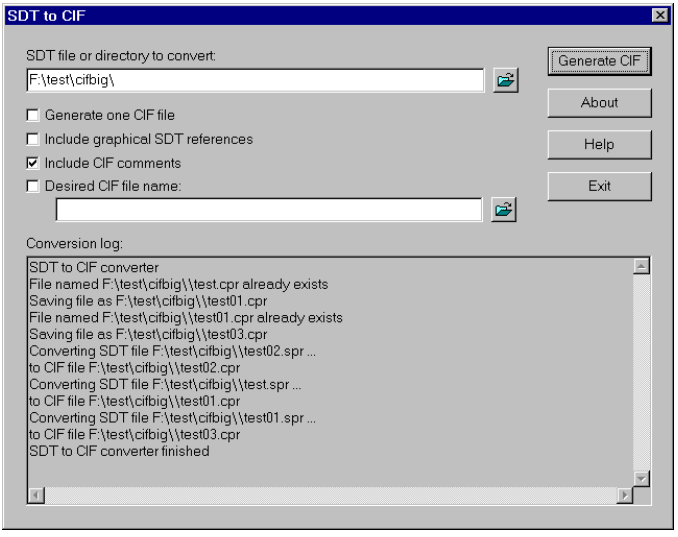


Figure 172: SDT2CIF Graphical User Interface

To convert a single binary file, system file or all binary files in a directory, perform the following steps:

1. Type an file or directory name in the text field under *file or directory to convert*.
   - To select a file using a standard dialog box, click the folder button which is located to the right of the text field.
2. Specify the desired options by clicking on check boxes which specify conversion options; look for option explanations in the following subsections.
3. Click the *Generate CIF* button.

After these steps, the conversion will proceed, possibly printing warning/errors/information messages into the *Conversion log* text box.

After the conversion has been performed, the SDT2CIF converter can be used again to convert more files or directories.

Converter Options

- *file or directory to convert*

  This text field specifies the name of the file or directory to be converted. To select a file using a standard file selection dialog box, press the folder button located to the right of the text field.

- *Generate one CIF file*

  If this option is enabled, the converter will save all converted diagrams into one file. The name of this file is determined from the *Desired CIF file name* check box state. If the check box is not selected, the name of the file will be the same as the name the first file being converted would get. This option is disabled by default. If the check box is selected, the name of the file with diagrams will be the name specified in the *Desired CIF file name* text field.

- *Include graphical SDT references*

  This option directs the SDT2CIF converter to include graphical SDT references in the generated CIF files (see chapter 18, *SDT References* for more information on this topic). If this option is enabled, graphical SDT references are stored as SDL/PR comments; otherwise they are omitted. This option is disabled by default.

- *Include CIF comments*

  If this option is disabled, only SDL/PR will be generated to output file; otherwise CIF comments will be included. This option is enabled by default.

- *Desired CIF file name*

  This text field specifies the name of the file that will contain the converted diagrams. To select a file using a standard file selection dialog box, press the folder button located to the right of the text field.

- *Conversion log*

  This text box contains warning/error/information messages issued by the converter during the last conversion. The possible messages are listed in "Messages from SDT2CIF Converter" on page 908.

- *Generate CIF*

  Clicking this button will initiate the conversion, using the options currently displayed in the dialog.

- *Exit*

  Clicking this button will exit the converter.

Messages from SDT2CIF Converter

The SDT2CIF converter issues an *information message* when it needs to inform the user about something; for example, when it changes the output file name so as not to overwrite an existing file. It also issues a *warning message* when some non-fatal error is found; for example, when some symbol on a diagram occurs in a wrong context. It prints an *error message* when it is not possible to continue the conversion; for example, when the input file is wrong or corrupt.

Message Format

The general format for warning/error messages is the following:

```
ERROR <error code>: <error text> <additional
information>
WARNING <warning code>: <warning text> <additional
information>
```

The `<error code>` specifies the error code which can be used to find the warning/error explanation (see the following sections). The `<error text>` gives a short explanation of what is wrong. The `<additional information>` specifies additional information about the error (for example, it can specify the name of an endpoint constraint that could not be bound).

The list of possible warning/error messages follows in the next sections.

List of Error Messages

`Error 1: Arguments required` **(UNIX only)**
This error indicates that no command-line arguments were given to the SDT2CIF converter, which thus cannot continue processing. To remedy the situation, supply one or more file/directory name(s) to convert.

`Error 2: Illegal option` **(UNIX only)**
This message is issued when an invalid option is found in the command line. To remedy the situation, supply an appropriate option instead of the invalid one.

`Error 3: Duplicate option` **(UNIX only)**
This message is issued when a duplicate command-line option is found. The command line options can be specified only once. To remedy the situation, remove duplicated options from the command line.

`Error 4: Missing output file` **(UNIX only)**
This message is issued when the option `-o` has been specified, but no output file name follows. To remedy the situation, supply an output file name after the `-o` option.

`Error 5: Illegal command line syntax` **(UNIX only)**
This message is issued when the command line is found not to obey the command-line syntax. To remedy the situation, make the command line conform to the command-line syntax.

`Error 6: Impossible to connect to SDT PostMaster`
This message is issued when the SDT2CIF converter cannot connect to the PostMaster.

In Windows, the SDT2CIF converter requires the PostMaster to be running in order to perform conversion. Start the Organizer and try again.

On UNIX, the most likely cause of this message is that either the path to the SDL Suite tools is not in the search path, or the maximum number of licenses is reached. To remedy the situation, ensure that the SDL Editor can be started (i.e. it is in the search path) and that there are enough licenses available.

Error 7: SDLE cannot convert the file

This message means that either the file is corrupt or the SDL Editor cannot read the source file or write to output file. To remedy the situation, ensure that the source file is valid and that there is enough disk space. Try to Restart the SDL Suite and re-run the converter.

List of Warning Messages

Warning 1: No input files (UNIX only)

This message means that the SDT2CIF converter did not find any suitable file for conversion among those specified on the command line. To remedy the situation, ensure the specified file/path names are correct.

Warning 2: Cannot convert file or directory

This message is most probably caused by a path/file name misspelling. To remedy the situation, ensure that the specified file/path names are correct.

Warning 3: SDLE says: ...

This warning is issued when the SDT2CIF converter receives reply from the SDL Editor with message explaining the reason of the error. To see the more detailed explanation of the reason for the error, see the text following the colon.

Chapter

17 *The Information Server*

This chapter is a reference to the error messages that are issued by the Information Server.

General

The Information Server is an application that serves the SDL Suite tools as a response to the following requests:

- Type Viewer request to list SDL diagram types along with their relationships and their instantiations. See chapter 45, *The SDL Type Viewer* for more information.
- SDL Editor request to list SDL signals according to the options selected in the Signal Dictionary options. See "Specifying the Signal Dictionary Options" on page 1986 in chapter 43, *Using the SDL Editor*.

The Information Server has no user interface and its operation is managed under the supervision of the SDL Suite, beyond user control.

Information Server Error Messages

This section is a reference to the error messages that may be produced by the Information Server.

Some messages contain a '#' followed by a number, which is used to indicate where information, specific to the error situation, will be included.

Some messages include a reference to the object that is the source of the diagnostic. These messages adhere to the format adopted in the SDL Suite. See chapter 18, *SDT References* for a reference to this format and for examples.

`ERROR 1000 #1 #2 contains parse error at '#3'`

The message contains a reference to an entity that could not be parsed by the Information Server. It is likely that the symbol contains syntactically incorrect information. The error message also contains a reference to the object where the error was detected.

`ERROR 1001 #1 #2 is not connected to a file`

#1 and #2 refer to the type and name of a diagram that is not connected in the Organizer structure. Use the Organizer command *Connect* (see "Connect" on page 92 in chapter 2, *The Organizer*) to connect the diagram to a suitable file.

ERROR #1002 Inheritance for #1 #2 is circular

#1 and #2 refer to the type and name of a diagram that contain a circular inheritance. Circular inheritance is not supported by the SDL Suite. The error message also contains a reference to the object where the error was detected.

ERROR 1003 Could not create temporary filename, status: #1

The Information Server failed when creating the name of a temporary file. #1 contains more information about what operation failed.

ERROR 1004 Could not create temporary file, status: #1

The Information Server failed when creating a temporary file. #1 provides additional information about the error code.

ERROR 1005 #1 #2 contains yacc stack overflow at '#3'

The message contains a reference to an entity which caused the parser stack to overflow. This message may occur when the parsers processes statements that contain a large amount of lexical elements. Break down the expression. The error message also contains a reference to the object where the error was detected.

Chapter 17 The Information Server

Chapter

18 *SDT References*

This chapter contains a reference to the syntax used by the SDL Suite when referring to entities managed by the SDL Suite.

Through these references, the SDL Suite environment provides a means to:

- **Obtain a graphical trace back from Analyzer diagnostics to the source SDL diagrams**
- **Trace simulations in graphical mode**
- **Obtain a trace back from generated MSCs to the source SDL diagrams**
- **Navigate between SDL type diagrams, cross-reference diagrams, coverage charts and the source SDL diagrams**
- **Navigate between the generated PR or C files and the source SDL diagrams.**

General

The concept “GR reference”, introduced in the SDT 2.X tool family, has been extended in order to support references to virtually all entities that are managed in an SDT 3.X environment (or will be managed in future versions):

- SDL diagrams
- SDL pages
- SDL symbols
- Text within an SDL symbol
- SDL Overview diagrams
- SDL/PR files
- OM, SC, MSC, HMSC diagrams
- OM, SC, HMSC pages
- OM, SC, MSC, HMSC symbols
- Text within an MSC symbol
- Text within an OM symbol[1]
- Cross-reference files
- Coverage files
- C files
- ASN.1 files
- Other ASCII text files

The term *GR reference* has been replaced by the term *SDT reference*. It is possible to retrieve SDT references from the SDL Suite tools using commands such as *Show GR Reference* in the SDL Editor, and to show the source that matches a reference using commands such as *Go To Source* in the Organizer.

1. This refers to the text in the name compartment in class definition and class instance symbols. Text in the operation and attributes compartments are not possible to refer to in this version.

Syntax

Below follows the syntax for an SDT reference:

```
<SDT_Reference> ::=
  '#SDTREF' '('
  ( <SDT_GR_Ref> | <SDT_Textfile_Ref>
  | <SDT_MSC_Ref> | <SDT_HMSC_Ref> | <SDT_OM_Ref>
  | <SDT_SC_Ref> )
  ')'

<SDT_GR_Ref> ::=
   'SDL' ',' <FileName>
   [ '(' <PageName> ')' ]
   [ ',' <ObjectId> [ <Object_Coordinates> ]
   [ ',' <LineNumber> [ ',' <Column> ] ] ]

<SDT_Textfile_Ref> ::=
  'TEXT' ',' <FileName>
   [ ',' <LineNumber> [ ',' <Column> ] ]

<SDT_MSC_Ref> ::=
   'MSC' ',' <FileName>
   [ ',' <ObjectId>
   [ ',' <LineNumber> [ ',' <Column> ] ] ]

<SDT_HMSC_Ref> ::=
   'HMSC' ',' <FileName>
   [ '(' <PageName> ')' ]
   [ ',' <ObjectId> [ <Object_Coordinates> ]
   [ ',' <LineNumber> [ ',' <Column> ] ] ]

<SDT_OM_Ref> ::=
   'OM' ',' <FileName>
   [ '(' <PageName> ')' ]
   [ ',' <ObjectId> [ <Object_Coordinates> ]
   [ ',' <LineNumber> [ ',' <Column> ] ] ]

<SDT_SC_Ref> ::=
   'SC' ',' <FileName>
   [ '(' <PageName> ')' ]
   [ ',' <ObjectId> [ <Object_Coordinates> ]
   [ ',' <LineNumber> [ ',' <Column> ] ] ]

<FileName> ::=
  File name

<PageName> ::=
  name of a page according to SDT 3.X rules

<ObjectId> ::=
  integer, unique id for object
```

```
<Object_Coordinates> =
  '(' <x-coord> ',' <y-coord> ')'

<x-coord> ::=
   x coordinates in mm from upper left corner
<y-coord> ::=
   y coordinates in mm from upper left corner

<LineNumber> ::=
   line number within symbol or file

<Column> ::=
   column within the line
```

Examples

Below follow some examples of SDT references.

Example 155: SDL graphical references ––––––––––––––––––––––

Reference	Explanation
`#SDTREF(SDL,test.sbk)`	SDL diagram.
`#SDTREF(SDL,test.sbk(P1))`	SDL page
`#SDTREF(SDL,test.sbk(P1),10)`	SDL symbol
`#SDTREF(SDL,test.sbk,10)`	SDL symbol
`#SDTREF(SDL,test.sbk(P1),10(30,130))`	SDL symbol
`#SDTREF(SDL,test.sbk(P1),10,5)`	Line within symbol
`#SDTREF(SDL,test.sbk(P1),10(30,130),5)`	Line within symbol
`#SDTREF(SDL,test.sbk(P1),10,5,15)`	Column at line within symbol

Example 156: MSC graphical references ––––––––––––––––––––––

Reference	Explanation
`#SDTREF(MSC,test.msc)`	MSC
`#SDTREF(MSC,test.msc,10)`	Symbol in MSC
`#SDTREF(MSC,test.msc,10,5)`	Line within symbol
`#SDTREF(MSC,test.msc,10,5,15)`	Column at line within symbol

Example 157: OM graphical references ––––––––––––––––––––––––

Reference	Explanation
`#SDTREF(OM,test.som)`	OM diagram.
`#SDTREF(OM,test.som(P1))`	OM page
`#SDTREF(OM,test.som(P1),10)`	OM symbol
`#SDTREF(OM,test.som,10(30,130))`	OM symbol
`#SDTREF(OM,test.som,10,5)`	Line within symbol
`#SDTREF(OM,test.som(P1),10(30,130),5)`	Line within symbol
`#SDTREF(OM,test.som(P1),10,5,15)`	Column at line within symbol

Example 158: Textual references ––––––––––––––––––––––––––––

Reference	Explanation
`#SDTREF(TEXT,test.pr)`	Text file
`#SDTREF(TEXT,test.pr,10)`	Line in text file
`#SDTREF(TEXT,test.pr,10,15)`	Column at line within file

––

Chapter

19 *Using OM Access*

This chapter describes the OM Access feature.

OM Access is a module that provides the end user with access to the information in a diagram created by the OM Editor through a C++-interface, based on the UML Meta-model.

OM Access

OM Access is a C++ application programmer's interface to the OM Editor. It can be used in applications that uses the information contained in OM diagrams, such as:

- Code generators
- Documentation generators
- Report generators
- Statistics.

Note:

This chapter is an OM Access primer and is not intended to provide knowledge about C++ programming.

OM Access Files

These files can be found under the `<installation dir>/orca/omaccess/>` directory of your installation.

Filename	Description
`stlmini.h`	Minimal implementation of the standard C++ library, used by `omaccess.cc`
`uml.h`	Declarations of UML Meta-structure
`omaccess.h`	Header-file for `omaccess.cc`
`omaccess.cc`	Functions for accessing the data in an OM diagram
`pmtool.h`	Declarations for low-level communications with the OM Editor
`om2cpp.cc`	A small demo program that takes a diagram and makes a C++ include file
`trace.cc`	A small demo program that prints almost everything it gets from OM Access

General Concepts

The OM Access Application

An OM Access application is an application that uses the data in an OM Diagram. This OM Diagram is represented in the program by a data structure similar to UML's metamodel.

The first section introduces the basic methods.

Basic Methods

First of all there are some definitions needed; these can be found in the include file `omaccess.h`:

```
#include "omaccess.h"
```

This file also includes `uml.h` and `stlmini.h`. The file `uml.h` contains declarations for the UML metamodel, and `stlmini.h`[1] contains a minimal implementation of `string` and `list` classes in the standard C++ library used for handling the data.

An instance of the `OMModule` is needed to hold the information:

```
OMModule module;
```

To load the contents of a diagram into the module either `GetFile` or `GetBufID` can be used:

```
bool result = GetFile(module, filename);
bool result = GetBufID(module, bufid);
```

Returns `true` on success, false if failed to load the module.

1. `stlmini.h` can be replaced by the appropriate standard C++ headers `<list.h>` and `<string.h>` if you have access to an implementation of the standard C++ library.

Accessing the Information

The `OMModule` class keeps the information fetched from the OM Editor in lists. The lists are:

```
list<Class> classList
list<Generalization> generalizationList
list<AssociationClass> associationlist
```

Aggregations and associations are stored into the `associationList`, to simplify handling. To pick out the associations and aggregations separately, use the functions:

```
GetAggregations(OMModule&, list<AssociationClass>*)
GetAssociations(OMModule&, list<AssociationClass>*)
```

These functions fill the list in the second parameter with the associations and aggregations in the module.

The lists can be traversed using iterators. The following example shows how to print all the names of the classes in a module:

Example 159 ----------

```
for (list<Class>::iterator ci=omModule.classList.begin();
     ci != omModule.classList.end();
     ++ci) {
        const Class &omClass = *ci;
        cout << omClass.name << endl;
}
```

Relations

The links between associations/aggregations/generalizations and classes are represented as the names of classes, stored as `strings`.

Generalizations

The `Generalization` class contains the data members `subtype` and `supertype`. The names of the super- and subclass are stored as `string`.

The data member `discriminator` contains the discriminator, stored as `string`.

To list the superclasses and subclasses of a class, the following functions can be used:

```
GetSuperClassList(OMModule&, string&, list<string>*)
GetSubClassList(OMModule&, string&, list<string>*)
```

They put the names of the superclasses or subclasses belonging to the name of the class in `string`, into the `list`.

Aggregations/Associations

The `AssociationClass` class can contain both aggregations and associations as well as associations with association classes connected to them. The `boolean` member `isAggregation` can be used to determine if it is an aggregation or association.

To get a list of the names of the classes connected to the association/aggregation, the function

```
GetEndPoints(AssociationClass&, list<string>*)
```

can be used.

Example

Here follows a small example that shows how to access the class information. For each class in the diagram it prints the name of the class and the name of its subclasses.

Example 160: printclasses.cc ----------

```
#include <iostream.h>
#include "omaccess.h"

#include "stlmini.h"

int main(int argc, char *argv[]) {
        if (argc !=2) {
                cout << "Usage : printclasses <filename>" << endl;
                return EXIT_FAILURE;
        }

        OMModule omModule;

        // Retrieve the file from the OM Editor
        if(! GetFile(argv[1], &omModule) )
                return EXIT_FAILURE;

        // Iterate over the classes
        for (list<Class>::iterator ci=omModule.classList.begin();
             ci != omModule.classList.end();
             ++ci) {
                const Class &omClass = *ci;
                cout << omClass.name << endl;

                // Extract the subclassess
                list<string> classlist;
                GetSubClassList(omModule,omClass.name,&classlist);
                // And iterate through them
                for (list<string>::iterator ni=classlist.begin();
                     ni != classlist.end();
                     ++ni) {
                        const string &name = *ni;
                        cout << " Subclass: " << name << endl;
                        }
        }
        return EXIT_SUCCESS;
}
```

Files and Compiling

General

To use OM Access a C++ compiler that can handle templates is needed, such as g++ 2.95.2, or MSVC 6.

There are some Makefiles together with the examples (`$telelogic/orca/omaccess/examples`) that might be usable as templates.

OM Access is based on the Public Interface; see "PostMaster Reference" on page 490 in chapter 10, *The PostMaster*.

UNIX

The following files should be included into the compilation/linking phase for UNIX is:

```
$telelogic/orca/omaccess/src/omaccess.cc
$telelogic/orca/omaccess/src/pmtool.cc
$telelogic/lib/<platform>lib/post.o
```

where `$telelogic` refers to the installation path, and `<platform>` can be one of `linux` or `sunos5`.

Two paths to the include files is also necessary:

```
-I$telelogic/orca/omaccess/include
-I$telelogic/include/post
```

Example 161: Compile `printclasses.cc` with g++ on Solaris ––––––––––––––––––

```
g++ -o printclasses -I$telelogic/orca/omaccess/include \
        -I$telelogic/include/post \
        printclasses.cc \
        $telelogic/orca/omaccess/src/omaccess.cc \
        $telelogic/orca/omaccess/src/pmtool.cc \
        $telelogic/lib/sunos5lib/post.o
```

––

If Solaris is used, the switches `-lgen -lsocket -lnsl` have to be added.

Example 162: Compile `printclasses.cc` with g++ on Solaris

```
g++ -o printclasses -I$telelogic/orca/omaccess/includes \
        -I$telelogic/include/post \
        printclasses.cc \
        $telelogic/orca/omaccess/src/omaccess.cc \
        $telelogic/orca/omaccess/src/pmtool.cc \
        $telelogic/lib/sunos5lib/post.o -lgen -lsocket -lnsl
```

The OM Access files could also be copied to a local directory to ease up compilation.

Windows

> **Note:**
>
> During runtime the DLL `post.dll` (located in a subdirectory to `<installation directory>\sdt\sdtdir\wini386\include`) must either be in the path or the directory the application is started from.

Microsoft Visual C++

Add the files `omaccess.cc, pmtool.cc` (can be found in `<installation directory>\orca\src`) and `post.lib` (can be found in `<installation directory>\sdt\sdtdir\wini386\include\msvc50)` to the project.

Add `<installation directory>\orca\includes` and `<installation directory>\include\post` to the include path.

Using OM Access Together with the SDL Suite

Applications written with OM Access can easily be called from within the OM Editor, for example from a menu (for more info about defining your own menus see *"Defining Menus in the SDL Suite" on page 18 in chapter 1, User Interface and Basic Operations*.

The following example adds a choice to generate C++ code skeleton of an OM Diagram from the OM Editor and pops up a text editor with the generated file in it.

Example 163: `ome-menus.ini` ----------

```
SDT-DYNAMICMENUS-3.6
[MENU]
Name=&OM Access
[MENUITEM]
ItemName=Generate C++
Separator=off
StatusbarText=Generate C++ code skeleton
ProprietaryKey=1
AttributeKey=0
Scope=Always
ConfirmText=You might want to change the name of the output file
ActionInterpretation=OS_COMMAND
BlockCommand=off
FormattedCommand=om2cpp %b `basename %b .som`.h
[MENUEND]
```

Reference

The reference contains two parts:

- Data Model: information about the data model used. For each class, a description is provided and then the members of the class are presented in a table.
- Functions: information about the functions in OM Access. For each function, the definition is shown followed by a description and a table of the parameters.

Data Model

The data model in OM Access is based on the UML Metamodel. For more information about the UML Metamodel, see the “UML Semantics” chapter of the “Unified Modeling Language, version 1.1” documentation. This document can be found at:
`http://www-306.ibm.com/software/rational/uml/`

If not mentioned otherwise, the members are of type `string`.

OMModule

An instance of the `OMModule` class contains information about an OM Diagram.

Member	Description
`classList`	A list of `Class`, containing all of the classes.
`generalizationList`	A list of `Generalization`, containing all of the generalizations.
`associationList`	A list of `AssocationClass`, containing all of the associations/aggregations.

Class

An instance of the `Class` class contains information about a class.

Member	Description
`name`	The class name as a string.
`attributeList`	A list of `Attribute`, each entry representing one attribute in the class.
`operationList`	A list of `Operation`, each entry representing one attribute in the class.

Operation

An instance of the `Operation` class contains information about an operation.

Member	Description
`visibility`	Visibility of operation.
`name`	Name of operation.
`returnType`	Return type of operation.
`parameterList`	List of `Parameter` containing information about the operations parameters.

Parameter

An instance of the `Parameter` class contains information about a parameter.

Member	Description
`name`	Name of the parameter.
`type`	Type of the parameter.

Attribute

An instance of the `Attribute` class contains information about an attribute.

Member	Description
`visibility`	Visibility of attribute.
`name`	Name of attribute.
`type`	Type of the attribute.
`value`	Default value.

Generalization

An instance of the `Generalization` class contains information about a generalization.

Member	Description
`subtype`	The name of the subtype, ‘subclass’ of the generalization.
`supertype`	The name of the supertype, ‘superclass’ of the generalization.
`discriminator`	The discriminator of the generalization.

AssociationClass

An instance of the `AssociationClass` class contains information about an association/aggregation. The connection between the different classes is maintained by `fromEnd` and `toEnd`.

Member	Description
`fromEnd`	An `AssociationEnd` representing one of the ends of the association.
`toEnd`	An `AssociationEnd` representing the other end of the association.

Member	Description
isAggregation	A boolean that is true if the association is an aggregation.
hasAssociationClass	A boolean that is true if the association also have an AssocationClass.

If the association also has an associationclass, the class-information is contained in the instance, inherited from Class.

AssociationEnd

An instance of the AssociationEnd class contains information about a connection to a class, for example in an association.

Member	Description
name	Name of this end.
rolename	Role name of association.
type	Name of the class connected to.
multiplicity	Multiplicity of association.
qualifier	Qualifier of association.
constraint	Constraint of association.
aggregation	Represents the type of the association.
isSorted	True if the association is sorted.
isOrdered	True if the association is ordered.

Functions

GetFile

```
bool GetFile(filename, omModule, status)
```

Description

Loads the contents of `filename` into `omModule`. Returns `true` on success, `false` if failed to load the module.

Parameters

Parameter	Type	Description
`filename`	`string`	The (absolute) name of the file containing the diagram.
`omModule`	`OMModule *`	A pointer to the OMModule where the diagrams data will be put.
`status`	`string *`	(optional) If loading failed a diagnostic message is inserted into this string.

GetBufID

```
bool GetBufID(bufID,omModule,status)
```

Description

Loads the contents of the buffer `bufID` into `omModule`. Returns `true` on success, `false` if failed to load the module.

Parameters

Parameter	Type	Description
`bufID`	`string`	The bufferID of the diagram.
`omModule`	`OMModule *`	A pointer to the OMModule where the diagrams data will be put.
`status`	`string *`	(optional) If loading failed a diagnostic message is inserted into this string.

GetSuperClassList

```
GetSuperClassList(omModule,className,superClasses)
```

Description

Fills the list `superClasses` with the names of the superclasses to `className` in `omModule`.

Parameters

Parameter	Type	Description
`omModule`	`OMModule &`	The OMModule containing the diagram information.
`className`	`string`	The name of the class whose superclasses are searched for.
`superClasses`	`list<string> *`	The list to put the name of the superclasses in.

GetSubClassList

```
GetSubClassList(omModule,className,superClasses)
```

Description

Fills the list `subClasses` with the names of the subclasses to `class-Name` in `omModule`.

Parameters

Parameter	Type	Description
`omModule`	`OMModule &`	The OMModule containing the diagram information.
`className`	`string`	The name of the class whose subclasses are searched for.
`subClasses`	`list<string> *`	The list to put the name of the subclasses in.

SourceAggregationEnd

```
string SourceAggregationEnd(association)
```

Description

Returns the name of the class which starts the aggregation (the end with the square in it), the owner-part. (The `source` contains the `end`.)

Parameters

Parameter	Type	Description
`association`	`AssociationClass`	The association to check.

TargetAggregationEnd

```
string TargetAggregationEnd(association)
```

Description

Returns the name of the class that ends the aggregation (the end without the square in it), the owned-part. (The `source` contains the `end`.)

Parameters

Parameter	Type	Description
`association`	`AssociationClass`	The association to check.

GetEndPoints

```
void GetEndPoints(association, classes)
```

Description

Fills the classes list with the names of the classes that this association/aggregation is connected to.

Parameters

Parameter	Type	Description
`association`	`AssociationClass`	The association to check.
`classes`	`list<string> *`	The list to insert the names into.

HasEndpoint

```
bool hasEndPoint(association, name)
```

Description

Returns true if `association` has an endpoint at `name`.

Parameters

Parameter	Type	Description
`association`	`AssociationClass`	The association to check.
`name`	`string`	The class to check.

GetAggregations

```
void GetAggregations(omModule,aggregations)
```

Description

Fills `aggregations` with all the aggregations in `omModule`.

Parameters

Parameter	Type	Description
`omModule`	`OMModule`	The OMModule containing the diagram information.
`aggregations`	`list<AssociationClass>`	The list to fill with aggregations.

GetAggregations

```
void GetAggregations(comModule,aggregations,owner)
```

Description

Fills `aggregations` with all the aggregations in `omModule`, that has `owner` as source.

Parameters

Parameter	Type	Description
`omModule`	`OMModule`	The OMModule containing the diagram information.
`aggregations`	`list<AssociationClass>`	The list to fill with aggregations.
`owner`	`string`	the name of the class that has to be the owner.

GetAssociations

```
void GetAssociations(omModule, associations)
```

Description

Fills `associations` with all the associations in `omModule`.

Parameters

Parameter	Type	Description
`omModule`	`OMModule`	The OMModule containing the diagram information.
`associations`	`list<AssociationClass>`	The list to fill with associations.

GetAssociations

```
void GetAssociations(omModule, associations,
                     firstEnd)
```

Description

Fills `associations` with all the associations in `omModule`, that is connected to `firstEnd`, in some end.

Parameters

Parameter	Type	Description
`omModule`	`OMModule`	The OMModule containing the diagram information.
`associations`	`list<AssociationClass>`	The list to fill with associations.
`firstEnd`	`string`	The name of one of the classes that the association has to be connected to.

GetAssociations

```
void GetAssociations(omModule,associations,
                     firstEnd,secondEnd)
```

Description

Fills `associations` with all the associations in `omModule`, that is connected to `firstEnd`, and `secondEnd`.

Parameters

Parameter	Type	Description
`omModule`	`OMModule`	The OMModule containing the diagram information.
`associations`	`list<AssociationClass>`	The list to fill with aggregations.

Parameter	Type	Description
firstEnd	string	The name of one of the classes that the association has to be connected to.
secondEnd	string	The name of the second class that the association has to be connected to.

TraceModule

```
void TraceModule(omModule,os)
```

Description

Outputs a textual dump of `omModule` to `os`, with as much information as possible, useful for debugging purposes.

Parameters

Parameter	Type	Description
omModule	OMModule	The OMModule containing the diagram information.
os	ostream	The stream to output the dump to.

Chapter

20

Basic Compiling Theory

This chapter gives the ground basis for basic compiling theory. It will give an introduction to compiling theory as it is the corner-stone on which TTCN Access is built upon. See chapter 21, *TTCN Access*.

This chapter is intended to be read by developers of executable test suites, translator developers and test result analyzers.

Basic Compiling Theory

A programming language can be defined by describing what its programs look like (the *syntax* of the language) and what its programs mean (the *semantics* of the language).

For specifying the syntax of a language, such as TTCN, the widely used notation called *context-free grammar*, or BNF (Backus-Naur Form) is used. (A more precise definition of context-free grammar will be defined in "Syntax Definition" on page 944).

To describe the semantics of a language is more difficult, as no appropriate notation for semantic description is available. Consequently, when specifying the semantics of a language an informal description technique has to be used. TTCN uses *Semantic actions* that, in natural language, describes the actual semantic given a specific context.

Besides specifying the syntax of a language, a context-free grammar can be used to help guide the translation of a programs. A grammar oriented compiling technique, known as syntax-directed translation, is very helpful for organizing a compiler front end.

The Compiler

Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another. Often, some of the phases may be grouped together and the intermediate representation between the grouped phases need not be explicitly constructed.

The first three phases of a compiler, forming the bulk of the analysis portion is often called *front end*. The front end consist of those phases that depend primarily on the source language and are largely independent of the target language. This front end normally includes lexical and syntactic analysis, the creation of the symbol table, semantic analysis and the generation of intermediate code.

The last phases of a compiler, the *back end*, include those portions of a compiler that depend on the target language. In the back end, one will find code optimization, code generation along with the necessary error handling and symbol-table operation.

This chapter will discuss and explain the concept of the front end thoroughly. The back end will be mentioned just for completeness. The reason for this is quite obvious:

TTCN Access is the front end to a TTCN compiler.

Lexical Analyzer

The lexical analyze is the first phase of a compiler. Its main task is to read input characters and produce as output a sequence of tokens that the parser uses for the next phase, the *syntax* analysis.

Since the lexical analyzer is the part of a compiler that reads the source text, it may also perform certain secondary tasks at the user interface. One such task is stripping out comments and white space in the form of blank, tab, and newline character. Another is correlating error messages from the compiler with the source program. For example, the lexical analyzer may keep track of the number of newline characters seen, so that a line number can be associated with an error message.

In lexical analysis the stream of characters making up the program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

Example 164 ––

For example, the characters in the assignment statement below:

```
position := initial + rate * 60
```

would be grouped into the following tokens:

1. The identifier `position`
2. The assignment symbol `:=`
3. The identifier `initial`
4. The plus sign
5. The identifier `rate`
6. The multiplicand sign
7. The number `60`

––

The blanks separating the characters of these tokens would normally be eliminated during lexical analysis.

Syntax Definition

In this section, a notation called a *context-free* grammar (grammar for short) is introduced. This notation is used for specifying the syntax of a language.

A grammar naturally describes the hierarchical structure of many programming language constructs. For example, an if-else statement in C has the form as below.

```
if ( expression ) statement else statement
```

That is, the statement is the concatenation of the keyword *if*, an opening parenthesis, an expression, a closing parenthesis, a statement, the keyword *else*, and another statement. Using the variables *expr* to denote an expression and the variables `stmt`, `stmt1` and `stmt2` to denote three (possible different) statements, this structuring rules can be expressed as

```
stmt -> if ( expr ) stmt1 else stmt2
```

in which the arrow may be read as “can have the form”. Such a rule is called a *production*. In a production lexical elements like the keyword **if** and the parentheses are called *tokens*. Variables like `expr` and `stmt2` represent sequence of tokens and are called *nonterminals*.

A *context-free grammar* has four components:

- A set of tokens, known as *terminal* symbols.
- A set of nonterminals.
- A set of productions where each production consists of a nonterminal, called the *left side* of the production, an arrow, and a sequence of tokens and/or nonterminals, called the *right side* of the production.
- A destination of one of the nonterminals as the *start* symbol.

Through this chapter, examples and pictures will be present in order to clarify a topic or explain a specific concept. As TTCN, at this stage, has a too large grammar definition to be included in this document, a language called SMALL is defined that in this chapter will be used for examples and discussions.

SMALL will be defined as a language only consisting of digits and the plus and minus signs, e.g., 9-5+2, 3-1 and 7. In the language SMALL, a plus or minus sign must appear between two digits, and expressions

as above will be referred to as “lists of digits separated by plus or minus sign”.

The following grammar specifies the syntax of this language with *list* as its start symbol:

```
1:list -> list + digit
2:list -> list - digit
3:list -> digit
4:digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

From production (3), a single digit by itself is a *list*. Productions (1) and (2) express the fact that if one take any *list* and follow it by a plus or a minus sign and then another digit one has a new *list*. For example, it is possible to deduce that 9-5+2 is a list as follows:

a) 9 is a list by production (3), since 9 is a digit.

b) 9-5 is a list by production (2), since 9 is a list and 5 is a digit.

c) 9-5+2 is a list by production (1), since 9-5 is a list and 2 is a digit.

This reasoning is illustrated by the tree in the next figure. Each node in the tree is labelled by a grammar symbol. An interior node and its children correspond to a production; the interior node corresponds to the left side of the production, the children to the right side. Such trees are called *parse trees* and are discussed in the next chapter.

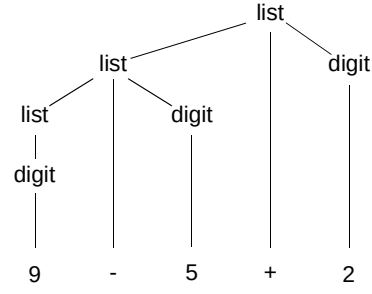


Figure 173: A parse tree for 9-5+2

Syntax Analyzer

Every programming language has rules that prescribe the syntactic structure of well-formed programs. In Pascal, for example, a program is made out of blocks, a block out of statement, a statement out of expressions, an expression out of tokens, and so on. The syntax of programming language constructs can be described by context-free grammars or BNF notation. Grammars offer significant advantages to both language designers and compiler writes:

- A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.
- From certain classes of grammars it is possible to automatically construct an efficient parser that determines if a source program is syntactically well formed.
- A properly designed grammar imparts a structure to a programming language that is useful for translation of source programs into correct object code and for the detection of errors.
- Languages evolve over a period of time, acquiring new constructs and performing additional task. These new constructs can be added to a language more easily when there is an existing implementation based on a grammatical description of the language.

During syntax analysis the tokens of the source program are grouped into grammatical phrases that are used by a compiler to synthesis the output. Usually, the grammatical phrase of the source program is represented by a *parse tree*.

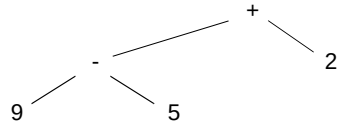


Figure 174: A syntax tree for the assign statement 9-5+2

Parse Trees

A parse tree pictorially shows how the start symbol of a grammar derives a string in the language. If nonterminal A has a production as:

```
A -> X Y Z
```

then a parse tree may have an interior node labelled A with three children labelled X, Y and Z, from left to right:

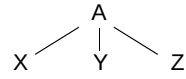


Figure 175: Parse tree

Formally, given a context-free grammar, a *parse tree* is a tree with the following properties:

1. The root is labelled by the start symbol.
2. Each leaf is labelled by a token or empty.
3. Each interior node is labelled by a nonterminal.
4. If A is the nonterminal labelling some interior node, and X1, X2, ..., Xn (where Xi are terminals or nonterminals) are the labels of the children of that node, then
   `A -> X1 X2 ..., Xn`
   is a production in the grammar.

The leaves of a parse tree read from left to right form the yield of the parse tree, which is the string *generated* or *derived* from the nonterminal at the root of the parse tree. Any tree imparts a natural left-to-right order to its leaves, based on the idea that if *a* and *b* are two children with the same parent, and *a* is to the left of *b*, then all descendants of *a* are to the left of descendants of *b*.

Another definition of a language generated by a grammar is as the set of strings that can be generated by some parse tree. The process of finding a parse tree for a given string of tokens is called *parsing* that string.

Parsing

Parsing is the process of determining if a string of tokens can be generated by a grammar. In discussing this topic, it is helpful to think of a parse tree being constructed, even though a compiler may not actually construct such a tree. However, a parser must be capable of constructing the tree, or else the translation cannot be guaranteed correct.

Intermediate Code

After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. This intermediate representation can be seen as a program for an abstract machine. In the context of the TTCN Suite and TTCN Access, the intermediate representation is the actual parse tree generated by the Analyzer. For more information see chapter 26, *Analyzing TTCN Documents (on UNIX)* and chapter 31, *Analyzing TTCN Documents (in Windows)*.

Code Generator

The final phase of a compiler is the generation of target code, that is transforming the intermediate code into something useful such as an executable. As TTCN Access is an application programmers interface towards the intermediate representation, the parse tree, it is the TTCN Access application that will define the actual transformation of the intermediate code. It can be an executable as well as a reporter, interpreter, etc.

The Phases of a Compiler

Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another. A typical decomposition of a compiler is shown in the next picture. In practice, some of the phases may be grouped together and the intermediate representation between the grouped phases need not be explicitly constructed.

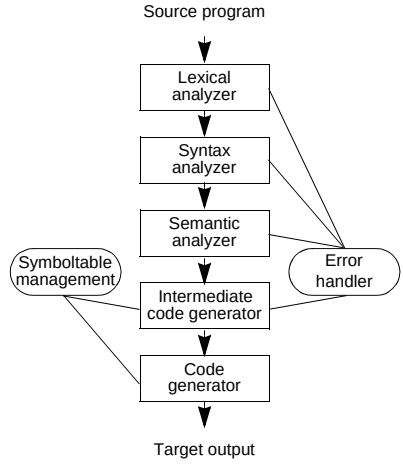


Figure 176: The phases of a compiler

The first three phases, forming the bulk of the analysis portion of a compiler has previously been discussed. Two other activities, symbol-table management and error handling, are shown interacting with the five phases of lexical analysis, syntax analysis, semantic analysis, intermediate code generation and code generation. Informally, the symbol-table management and the error handler are also called *phases*.

Symbol Table Management

An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier. These attributes may provide information about the identifier as its type, its scope (where in the source program it is valid) and, in the case of procedure names, such things as the number and type of its arguments, the method for passing each argument (e.g., by reference), and the type returned, if any.

A *symbol table* is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for a specific identifier quickly and to store or retrieve data from that record.

When an identifier in the source program is detected by the lexical analysis, the identifier is entered into the symbol table. However, the attributes of an identifier cannot normally be determined during lexical analysis. The remaining phases enter information about identifiers into the symbol table and then use this information in various ways. For example, when doing semantic analysis, the compiler needs to know what types the identifiers have, so it can be checked that the source program uses them in a valid way.

Error Detection and Reporting

Each phase can encounter errors. However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.

The syntax and semantic analysis phase usually handle a large fraction of the errors detectable by the compiler. The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operations involved, e.g., when adding two identifiers, one of which is the name of an array and the other is a name of a procedure.

Chapter

21 *TTCN Access*

This chapter describes TTCN Access. It is intended to be read by developers of executable test suites, translator developers and test result analyzers. It requires some basic programming knowledge in C++ as TTCN Access is a C++ programming tool.

Introduction to TTCN Access

A TTCN test suite can be looked upon as a formal description of a collection of test sequences where each test sequence involves signals and values. The abstract test suite contains formal definitions of these signals and values as well as a structural description of each test sequence. A common formal notation used to describe these test sequences, as well as all the other items in an abstract test suite, is TTCN.

TTCN Access is a C++ application programmers interface towards an arbitrary abstract test suite written in TTCN and incorporated in the TTCN Suite. TTCN Access reveals the content of the test suite in a high level abstraction and allows various users to access the content in a read-only manner.

TTCN Access is a platform for writing applications related to an abstract test suite such as:

- Executable test suites
- Interpreters
- Encoders and decoders
- Analyzers
- Reporters

By using TTCN Access, a variety of applications can be implemented that will ease the maintenance of abstract test suites and the development of executable test suites. TTCN Access is an easy-to-use application programmers interface that provides the required functionality for applications in these areas.

Terms Used in This Document:

- ISO/IEC 9646-3 : 1991 is called *the TTCN standard*.
- ISO/IEC 8824 : 1990 is called *the ASN.1 standard*.
- Backus-Naur Format is called *BNF*.

General Concepts

The easiest, and simplest way of describing TTCN Access would be to state that TTCN Access is a TTCN compiler. Unfortunately this statement is not fully true and also somewhat misleading as it might, conceptually, bring the reader (and TTCN Access users) towards the domain of executable test suites and thereby not reveal all the other possibilities that TTCN Access brings. A more correct statement would thereby be

that TTCN Access is a half compiler, more precisely the first phase of a compiler, often called the *front-end*. For a general explanation of compiler theory and compiler front-ends, see chapter 20, *Basic Compiling Theory*. The fact is that the TTCN to C compiler is built on top of TTCN Access.

TTCN Access and the TTCN Analyzer

All the components and definitions mentioned in chapter 20, *Basic Compiling Theory*, together build up the basics for compiling theory. As mentioned, TTCN Access is a C++ application programmers interface towards a test suite written in TTCN. It reveals the content of the test suite in a high level abstraction and allows various users to access the content in a read-only manner.

This section will once again mention these components, but this time put them in the context of the TTCN Suite and the functionality that it provides, thereby explaining the relationship between the TTCN Suite and TTCN Access.

Lexical Analysis

The lexical analysis in the TTCN Suite is done in two phases, depending on the format of the abstract test suite.

- If the source code is in MP format, a basic lexical analysis is done at the import stage, verifying that the imported test suite is written in correct MP format. A full lexical analysis is done when using the Analyzer. The complete lexical analysis is the first phase in the analysis.

  For more information see "Importing a TTCN-MP Document" on page 1157 in chapter 24, *The TTCN Browser (on UNIX)*.

- If the source code is written directly into the TTCN Suite via one of the editors, a lexical analysis is done by using the Analyzer as mentioned above.

  For more information see chapter 26, *Analyzing TTCN Documents (on UNIX)* and chapter 31, *Analyzing TTCN Documents (in Windows)*.

Syntax Analysis

The syntax analysis is done when executing the Analyzer and it is done in the second phase. This second phase also contains a semantic analysis of the test suite, all in order to verify that the test suite complies with the standardized notations for TTCN and ASN.1.

Parse Tree

The last phase of the analysis is to generate a parse tree. This can only be done if the lexical and syntax analysis have been successfully completed.

Symbol Table Management

During execution of an TTCN Access application the symbol table is accessible at any time.

Example of TTCN Access Functionality

As TTCN Access looks upon a test suite as a parse tree, it provides the required primitives and functionality for parse tree handling. This includes parse tree traversing and hooks into the parse tree as well as templates for main() and several examples in source code format. The best way to visualize the functionality of TTCN Access is by using an example:

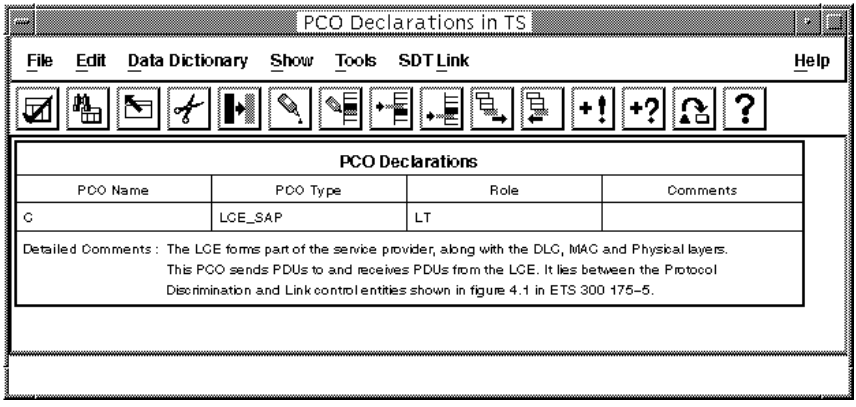


Figure 177: The PCO Declarations table

The PCO Declarations table, containing just one PCO declaration will generate the parse tree visualized below:

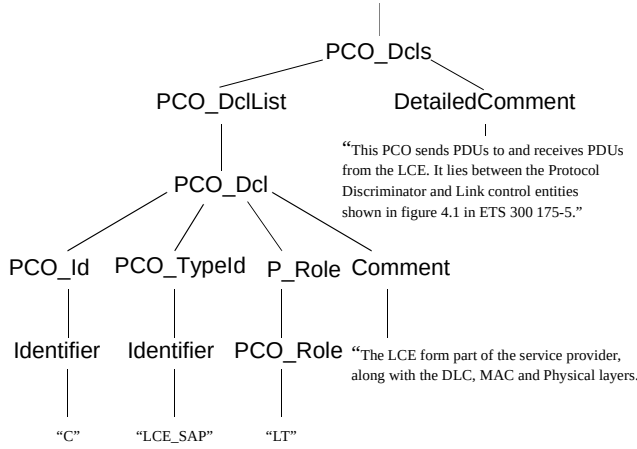


Figure 178: Parse tree

Traversing

The basics of parse tree functionality has to include traversing primitives. TTCN Access therefore provides a special *visitor* class with a *default traverser* that, given any node in the parse tree, traverses that parse tree in a *Left-Right-Depth-First* manner. In the parse tree previously described, the default traversing order for the sub tree *PCO_Dcl* will be:

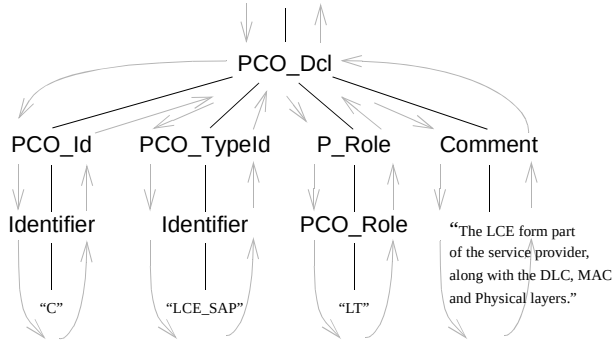


Figure 179: Default traversing tree

This default translating algorithm may be *customized* in order to fit any user specific traversing order. For the above example, a customized traverser could be implemented as traversing sub tree *PCO_TypeId* before sub tree *PCO_Id* and ignoring sub tree *Comment*.

Translating

The second most important functionality when discussing parse trees, is the possibility to access specific nodes in the parse tree in order to generate side effects, such as executing external code, print parse tree information, etc. By being able to modify the default traverser, TTCN Access provides the user with the possibility of defining such side effects.

Example of TTCN Access Usability

To visualize the strength of TTCN Access, this section will discuss a small TTCN Access application. The example is a generic encoder.

The Encoder

Transforming an abstract test suite into an executable test suite will eventually involve the problem of representing the actual ASP and PDU signals as bit patterns. The step from abstract syntax (TTCN) to transfer syntax (bit patterns) has to be provided and implemented by someone with vast knowledge of the actual protocol as well as the test system and the test environment.

The encoder described in this small example will assume the following:

- Elements of the same type are encoded identically.
- Encoding an ASP or PDU is performed by encoding its components recursively.

These assumptions tell us that encoding functions are data driven and that the encoding function has to be derived from the type definitions. It also tells us that the encoding of base types, that is INTEGER, BOOLEAN, etc., are identical.

Before looking at the algorithm for the generic encoder, let us have a look at a specific type and how the corresponding encoding function may be implemented. The type will be a small PDU as below:

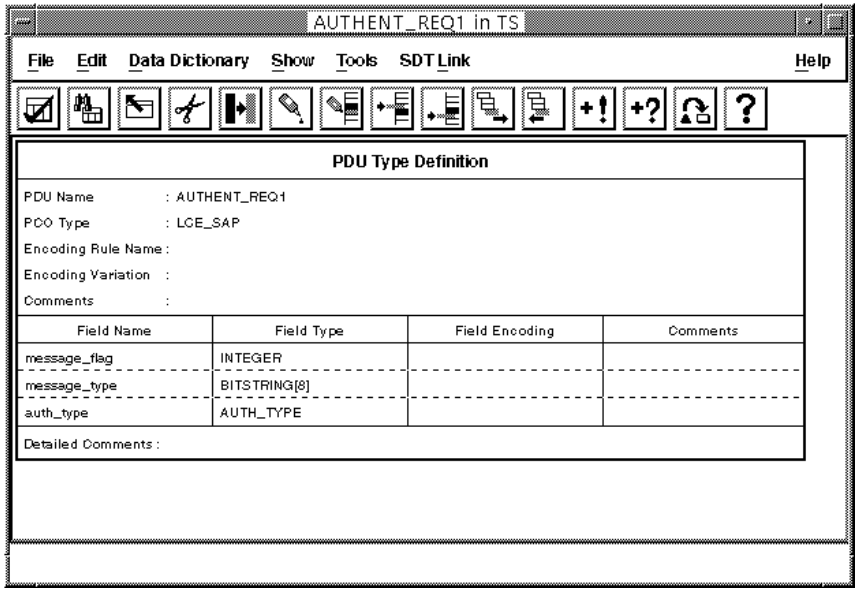


Figure 180: TTCN PDU type definition

The corresponding encoding function may look like:

```
void AUTHENT_REQ1_encode( AUTHENT_REQ1* me, char* enc_buf )
{
  INTEGER_encode( &me->message_flag, enc_buf );
  BITSTRING_encode( &me->message_type, enc_buf );
  AUTH_TYPE_encode( &me->auth_type, enc_buf );
}
```

It is a data driven encoding function that given any element of type `AUTHENT_REQ1` will encode them all identical. The first argument is a pointer to the element that shall be encoded, the second argument is the buffer where the result of the encoding shall be stored.

The function may of course contain more code and more information, but for this example, the above is the smallest encoder function needed.

The following example will give an algorithm for how the encode functions may be generated via TTCN Access. The algorithm is followed by a small TTCN Access application generating the actual encoder.

```
FUNCTION GenerateEncoder( type : typedefinition )
BEGIN
/* Generate encoder function header such as: */
/* void type_encode( type* me, char* enc_buf ) */
/* { */
FOR( "every element in the structured type" )
/* Generate a call to the element type encoder
/* function
/* such as: */
/* element_type_encode( &type->element, enc_buf );
END
/* Generate encoder function footer such as: */
/* } */
END
```

The above algorithm will generate generic encoders for any type. Of course, header files and base type encoder functions must be provided/generated as well. Below is a nearly complete TTCN Access application that generates encoders for any TTCN PDU type definition. Observe how the visitor class is customized to traverse and generate side effects for the specific parts we are interested in.

```
// Start by customizing the default visitor class

class Trav : public AccessVisitor
{
public:
  void VisitTTCN_PDU_TypeDef( const TTCN_PDU_TypeDef& Me );
  void VisitPDU_FieldDcl( const PDU_FieldDcl& Me );
};

void Trav::VisitTTCN_PDU_TypeDef( const TTCN_PDU_TypeDef& Me )
{
  // Generate the header part

  cout << Me.pdu_Identifier( ) << "_encode( const ";
  cout << Me.pdu_Identifier( ) << "& me, char* enc_buf )";
  cout << "\n{" << endl;

  // Traverse fields using default traverser

  AccessVisitor::VisitPDU_FieldDcls( Me.pdu_FieldDcls( ) );

  // Generate the footer part

  cout << "}" << endl;
}

void Trav::VisitPDU_FieldDcl( const PDU_FieldDcl& Me )
{
  Astring type = get_type( Me.pdu_FieldType( ) );
  Astring name = get_name( Me.pdu_FieldId( ) );

  cout << "void " << type << "_encode( me->"
       << name << "( ), enc_buf );" << endl;
}
```

All that remains now is a main function for the generic encode generator.

```
#include     <stdio.h>
#include     <iostream.h>
#include     <time.h>
#include     <ITEXAccessClasses.hh>
#include     <ITEXAccessVisitor.hh>

class Trav : public AccessVisitor
{
public:
  void VisitTTCN_PDU_TypeDef( const TTCN_PDU_TypeDef& Me );
  void VisitPDU_FieldDcl( const PDU_FieldDcl& Me );
};

int main( int argc, char **argv )
{
  //The actual suite

  AccessSuite suite;

  if ( argc != 2 )
    {
      fprintf( stderr, "usage: %s suitename\n", argv[ 0 ] );
      return 0;
    }

  /* Open the suite and go for it! */

  if ( suite.open( argv[ 1 ] ) )
    {
      Trav trav;

      trav.Visit( suite );
      suite.close( );
    }
}
```

The `get_type` and `get_name` functions can be simple or complex. See the example below:

```
Astring get_type( const PDU_FieldType& Me )
{
  TypeAndAttributes ta = Me.typeAndAttributes( );
  switch ( ta.choice( ) )
    {
    case Choices::c_TypeAndLengthAttribute:
      return get_type( ta.typeAndLengthAttribute( ) );
    default:
      cerr << "ERROR: Type not supported: "
           << Me.content( ) << endl;
      return "***#***";
    }
}


Astring get_type( const TypeAndLengthAttribute& Me )
{
  const TTCN_Type tt = Me.ttcn_Type( );

  switch ( tt.choice( ) )
    {
```

```
    case Choices::c_PredefinedType:
      if ( tt.predefinedType( ).choice( ) ==
           Choices::c_INTEGER )
        return "INTEGER";
      break;
    case Choices::c_ReferenceType:
      return tt.referenceType( ).identifier( );
    default:
      break;
    }
  cerr << "ERROR: Type not supported: "
       << Me.content( ) << endl;
  return "***#***";
}


Astring get_name( const PDU_FieldId& Me )
{
  const PDU_FieldIdOrMacro pfid = Me.pdu_FieldIdOrMacro( );
  if ( pfid.choice( ) != Choices::c_PDU_FieldIdAndFullId )
    {
      cerr << "ERROR: Slot name not supported: "
           << Me.content( ) << endl;
      return "***#***";
    }

  return pfid->pdu_FieldIdAndFullId( ).pdu_FieldIdentifier( );
}
```

TTCN Access in Relation to TTCN and ASN.1

TTCN Access is created for the purpose of accessing and traversing the contents of a TTCN test suite. As a basis for the development of TTCN Access, the TTCN-MP syntax productions in BNF [ISO/IEC 9646-3, appendix A] together with the ASN.1 standard [ISO/IEC 8824] have been used.

Differences

Every node in TTCN Access reflects a rule in the BNF. However there are some small differences between the BNF and TTCN Access:

The Present Nodes

The intention is to map the ASN.1 and TTCN standards as good as possible. In a perfect world this would mean that no extra nodes could be found in TTCN Access and that each rule in the standards would have exactly one corresponding node in TTCN Access. Some differences between the perfect world and our world are listed in detail below. Differences exist due to redundancies in the standardized grammar or design decisions made during the development of TTCN Access. The goal of any implementation of TTCN Access is to make as few changes compared to the perfect world as possible. However, some extra nodes have been added to TTCN Access, and in a few places the exact calling order of the pre/post functions has been altered to what we feel is a more straight forward approach for executable languages.

The SEQUENCE OF

The TTCN and ASN.1 notations at times allow for arbitrary many nodes to be grouped in lists of nodes or sequences of nodes. For example there is the SubTypeValueSetList rule of the ASN.1 standard and there is the AssignmentList rule of the TTCN standard. In both BNF notations a difference is made between lists that are allowed to be empty and those which must contain at least one element, corresponding to the {} and {}+ notation in the TTCN standard.

In TTCN Access no difference is made between the two forms of lists, since a well defined and simple access method is of prime interest. All lists are allowed to be empty, and it is the work of the Analyzer to ensure

non-empty lists where appropriate. In the ASN.1 standard the non-empty lists are sometimes written by re-grouping already existing rules. This re-grouped structure will not be represented in TTCN Access. The decision does not alter the number of nodes or the names of nodes, but merely the visiting order when traversing the nodes, i.e. SubTypeValueSetList before SubTypeValueSet.

Example 165 ----------

ASN.1 rule SubTypeSpec:

```
SubTypeSpec ::= ( SubTypeValueSet SubTypeValueSetList )
```

becomes:

```
SubTypeSpec ::= ( SubTypeValueSetList )
```

In the TTCN standard there are nodes which contain an implicit grouping.

Example 166 ----------

TTCN BNF rule:

```
TestStepLibrary ::= ({TestStepGroup | TestStep})
```

becomes:

```
TS_ConstDcls ::= {TS_ConstDcl}+ [Comment].
```

Furthermore there are nodes that contain groupings without the list postfix name convention.

Example 167 ----------

TTCN BNF rule:

```
TimerOps ::= TimerOp {Comma TimerOp}.
```

Whenever necessary, a grouping or/and choice node will be inserted between rules in the TTCN standard. When nodes are created without corresponding node in any of the standards, the nodes will be named according to the convention used in the ASN.1 standard, i.e. these nodes will have postfix “List”. Nodes which are a grouping but do not follow the naming convention will be left as they are.

OPTIONAL

In the BNF rules for TTCN-MP, the use of `[ abc ]` implies zero or one instance of `abc`. In the definition of TTCN Access the same effect is achieved by the use of the symbol “OPTIONAL” after the TTCN Access definition. If a TTCN Access element defined as “OPTIONAL” is not present in a specific instance, no TTCN Access representation for that specific field will be available. To detect whether or not a TTCN Access node defined as “OPTIONAL” is present or not, a boolean TTCN Access function will be necessary to apply to that TTCN Access node in order to verify the presence/absence of the parse tree related to the specific slotname.

FIELD

A difference between TTCN Access and the BNF rules is that TTCN Access views every BNF production reflecting a field in the TTCN-GR format to be optional even though the field is not defined as such in the BNF. The reason for this is that no TTCN Access tree can be built for a field unless a successful analysis has been performed on that field. If the analysis failed, the field will not carry any TTCN Access information and the field may not be accessed.

In TTCN Access all fields are marked with the symbol “FIELD”. This “FIELD” symbol implies that the field may be empty. The field may be empty even though the field is not defined as optional in the BNF. To detect whether or not a field is present or not present, a boolean TTCN Access function will be necessary to apply to that TTCN Access node in order to verify the presence/absence of the parse tree related to the specific field. This extension is applicable to all fields except if a field is implemented in TTCN Access as a node of type TERMINAL. All TERMINAL types are strings and TTCN Access sees the absence of a TERMINAL as an empty string.

The Value Notation

The Common Value Notation

The value notations of the two standards must be unified. Some values are clearly within one standard only (SetValue for instance), while others are in both standards (7 for instance). Those values that have the same syntax in both standards are considered to belong to the TTCN standard. Note that the ASN.1 standard might specify some other chain

of productions to reach such a value and that this chain of productions is not represented in TTCN Access.

The Expression Tree

The expression rule has been re-written since it is undesirable to have the flat representation from the standard. Instead a normal tree oriented representation of an expression is used. This means that the rules making up an expression are heavily changed.

The Base Nodes

Base nodes are the representation of the terminals in the TTCN Access tree. They can in most aspects be treated as strings, they can be printed directly for example. The base nodes are:

- Identifier
  Has an associated type
- Number
- All nodes having a single child of type BoundedFreeText (FullIdentifier or SO_SelExprId for example).
- Ostring, Cstring, Bstring and Hstring

Note:

The Keyword class in ITEX Access 1.0 does not exist in ITEX Access 2.0. Neither of the standards has the notion of a keyword class/rule, and therefore there is no node/class named Keyword in ITEX Access 2.0. It has been replaced with ITEX Access node TERMINAL.

Tree Traversing in the Dynamic Part

An extension to the BNF adds the possibility to access the content of test cases, test steps and defaults, by using the logical tree structure of the test. This is achieved by adding a slot in the node type definition BehaviourLine named “children”. By accessing that slot an TTCN Access object is returned that holds a vector of children to the current statement line. A child is a BehaviourLine with a level of indentation one greater that the preceding BehaviourLine.

Naming Conventions

As a basis for naming TTCN Access nodes, the names in the BNF for TTCN and ASN.1 have been used. There are some differences though, and they will be discussed and explained in this subsection.

- No slot name, type name or type definition contains the symbol “&” as it is a reserved symbol in C++. It is replaced with “and” or “And” where appropriate.
- All type definitions start with an upper case letter (and are written in bold).
- All slot names start with a lower case letter.
- All type names start with an upper case letter.

The TTCN Access Notation

The notation used to describe a node in TTCN Access is described as below (a simple BNF for an example node):

```
Node           ::= TypeReference Assign TypeAssignment
TypeReference  ::= Identifier
             --has to start with an upper case letter
Assign         ::= "::="
TypeAssignment ::= ClassType ClassBody | "TERMINAL"
ClassType      ::= "SEQUENCE" | "SEQUENCE OF" |
"CHOICE"
ClassBody      ::= "{" { Slot }+ "}"
Slot           ::= SlotName TypeReference
                   [ "OPTIONAL" | "FIELD"]
SlotName       ::= Identifier
              --has to start with a lower case letter
```

Example 168 ——————————————————————————————

```
StructTypeDef ::= SEQUENCE {
    structId          FullIdentifier  FIELD
    comment           Comment         FIELD
    elemDcls          ElemDcls
    detailedComment   DetailedComment FIELD
}
```

——

- A TTCN Access node type is defined in **Bold** starting with an upper case letter.
- A node can be of type SEQUENCE, SEQUENCE OF or CHOICE. A SEQUENCE contains an ordered collection of elements, a SE-

QUENCE OF a vector of elements (possibly empty) and a CHOICE is a collection of possible elements.

- Each slot in a *TypeAssignment* starts with a *SlotName* followed by a *SlotType*. Each *SlotType* has a corresponding node in TTCN Access.
- The *SlotType* can be followed by a `FIELD` symbol meaning that the slot is associated with a field in the TTCN-GR format.
- The *SlotType* can be followed by a `OPTIONAL` symbol meaning that the slot can be absent.
- Each slot is followed by a page reference to the node corresponding to the *SlotType*. This page reference is not a part of the Abstract Data Structure.

For more information see chapter 23, *The TTCN Access Class Reference Manual*.

TTCN Access Primitives

Each node in TTCN Access is translated to a C++ class definition. Each instance of a specific C++ class definition contains one or more elements as defined in the specific C++ class definition. For each type definition there are a collection of TTCN Access functions.

In this chapter names written in *italic* are referred to as meta names. Words written in `courier` are taken from the definition of TTCN Access.

> **Note:**
>
> For further details see the TTCN Access include file `access.hh`.

General TTCN Access Functions Description

- For every node definition with assignment of type SEQUENCE or SEQUENCE OF there exists a general method

  SlotType TypeReference.SlotName()

  that returns an object of type *SlotType*. This type is the type of the corresponding slot in TTCN Access.

  Example 169 ––

  ```
  Attach MyRepeat.attach()
  ```

  where MyRepeat is of type Repeat and the return value is of type Attach.

  ––

- For every *TypeAssignment* of type CHOICE there is a general method

  Choices::choice TypeReference.choice()

  that returns the allowed SlotName type for current object. It is then possible to use the general method

  SlotType TypeReference.SlotName()

  that returns an object of type SlotType.

  Example 170 ––

```
switch ( MyEvent.choice() ) {
        case Choice::c_send:
                do_something( Me.send() );
                break;
        case Choice::c_receive:
                do_something( Me.receive() );
                break;
        case Choice::c_otherwise:
                do_something( Me.otherwise() );
                break;
        case Choice::c_timeout:
                do_something( Me.timeout() );
                break;
        case Choice::c_done:
                do_something( Me.done() );
                break;
        default:
                do_something_default();
                break;
}
```

––

- For every node of type SEQUENCE OF there is a general method

  SlotTypeList TypeReference.SlotName()
  that returns an object of type *SlotTypeList* which is a vector of elements of type *SlotType*.

  Example 171 ----------

  Assume that MyEvent.choice() in the previous example returned the value Choice::c_done. Then, TTCN Access method MyEvent.done().tcompIdList will then be a valid method and return an object of type TCompIdList.

- Every object of type *SlotTypeList* has a method

  int *Object*.nr_of_items()
  that returns the number of elements in the vector. The elements can be accessed with the method

  SlotType Object[index]
  where *Object* is an element of type *SlotTypeList* and index starts with 0 for the first element. It returns an element of type *SlotType.*

  Example 172 ----------

  According to the above example the method MyTCompIdList.nr_of_items() returns an integer value of the number if items in the vector object MyTCompIdList and MyTCompIdList[2] will return the third element in the vector.

- For slots with ending OPTIONAL or FIELD there is a general method

  Boolean *TypeReference*.is_present_*SlotName*()
  used for verifying if an object is present or not. It is the responsibility of the TTCN Access programmer to ensure that all calls to optional slots are preceded with a call to the is_present method of that slot. It is a fatal error to attempt to access a non-present optional slot.

- For slots with ending `FIELD` and slots residing in a sub tree to a field there is a general method

  `Astring` *`Node`*`.content()`

  that returns an object of type `Astring` holding the content of the current field in a vector of characters.

Example 173 ––

The following C++ code will print out the content of a behavior line:

```
BehaviourLine BLine = MyLine;

  cout << BLine.line( ).content() << endl;
```

––

Terminal Nodes in TTCN Access

Every TTCN Access node that represents a leaf in the parse tree is considered to be a *terminal* TTCN Access node. A terminal TTCN Access node is a node that does not have any TTCN Access child nodes and can therefore not be accessed any further.

Terminal TTCN Access children are nodes containing identifiers, strings, keywords as well as numbers (i.e. TTCN Access nodes of type `Identifier`, `IA5String`, `INTEGER`, `NUMBER`, etc.). However, the content of such terminal nodes can be accessed through the methods supplied by the inherited class `Astring`.

TTCN Access Class Astring

Every terminal TTCN Access node carries information in string format. In order to access that information, every terminal TTCN Access node inherits a class named `Astring`. This class contains various methods for treating string information.

For terminal TTCN Access nodes, the following operations are available:

- Initialization operations

Example 174 ----------

The following C++ code will create and initiate a variable of type `Astring`:

```
Astring tmp1;              // tmp1 is empty
Astring tmp2( "test1" );   // tmp2 contains "test1"
Astring tmp3 = "test2";    // tmp3 contains "test2"
Astring tmp4 = tmp2;       // tmp4 contains "test1"
```

- Relational operations

Example 175 ----------

The following relational operations are available between Astring elements:

```
==, !=
```

- `Astring` objects are type cast equivalent with `const char*`

Example 176 ----------

The following code is valid C++ code:

```
Astring myString( "test" );

cout << myString;
printf( "%s", ( const char* ) myString );
```

- Address the nth character in the string, starting with 0.

Example 177 ----------

The following is valid C++ code:

```
char c = myString[ 2 ]; // save the third character
of the string
```

Note:

For further details see class `Astring` in TTCN Access include file `ITEXAccessClasses.hh`.

Direct Access

AccessSuite

The `AccessSuite` object is the TTCN Access representation of a TTCN test suite. The test suite is in turn contained in a TTCN Suite data base. The *AccessSuite* services include opening and closing the TTCN Suite data bases as well as services to start an TTCN Access application and accessing the symbol table manager.

Example 178 ––

Opening and closing the test suite *Test.itex* for use in TTCN Access:

```
AccessSuite suite;

Boolean ok_open = suite.open( "Test.itex" );
if( ok_open )
{
   // do something
   Boolean ok_close = suite.close();
}
```

––

After opening a test suite for TTCN Access use, it is possible to use the following methods to get a handle to the contents of the data base file:

- Get a handle to the root node of the document.

  Example 179 ––

  Getting a handle to the root object of a document:

```
AccessSuite suite;
Boolean ok_open = suite.open( "Test.itex" );
if( ok_open ) {
    const AccessNode node = suite.root();
    // do something with NSAPaddr
    Boolean ok_close = suite.close();
}
```

  ––

> **Note:**
>
> For further details see class `AccessSuite` in TTCN Access include file `access.hh`.

- Ask the AccessSuite object for a specific TTCN Access object in the test suite. The object asked for must be a global object in the test suite. This is performed by using TTCN Access class AccessNode.

Example 180 ----------

Find the TTCN Access object NSAP in test suite *Test.itex*:

```
AccessSuite suite;
Boolean ok_open = suite.open( "Test.itex" );
if( ok_open ) {
    const AccessNode node = suite.find("NSAPaddr");
    // do something with NSAPaddr
    Boolean ok_close = suite.close();
}
```

The AccessNode object now holds the TTCN Access item corresponding to the name NSAPaddr, if any.

To gain access to the data in an AccessNode, you must now find out the runtime type of the object. Based on that type, you will be able to use the conversion routine for an object of the corresponding type:

Example 181 ----------

Find the type of the TTCN Access node corresponding to a name:

```
extern void HandleSimpleType( const SimpleType * );

AccessSuite suite;
Boolean ok_open = suite.open( "Test.itex" );
if( ok_open ) {
   AccessNode node = suite.find( "SomeName" );
   switch( node.choice() ) {
     case Choice::_SimpleType:
       HandleSimpleType(node.SimpleType());
       break;
     default:
       break;
   }}
```

Note:

For further details see class AccessNode in TTCN Access include file access.hh

The AccessVisitor Class

TTCN Access is a large class library, with over 600 classes, and therefore we also need suitable tools for simplifying the creation of TTCN Access applications. The preferred solution is the usage of the class AccessVisitor, which definition is available in the C++ header file `ITEXAccessVisitor.hh`.

The AccessVisitor class is a very close relative to the design pattern 'Visitor' (described by, for instance, Gamma, Helm, Johnson and Vlissides in 'Design Patterns - Elements of reusable software', Addison-Wesley1994). The difference is that the TTCN Access classes contain runtime type information which eliminates the need for them to have dependencies to the Visitor class (and therefore there is no need for 'Accept' methods in the visited classes).

An object of a class which is derived from the class AccessVisitor is later on referred to as a 'visitor'.

AccessVisitor Class Members

Common Classes

The AccessVisitor class has two methods for visiting objects of common classes AccessSuite and AccessNode. These are declared

```
public:
void Visit( const AccessNode );
void Visit( const AccessSuite & );
```

and calling them with, will start a chain of calls in the visitor which effectively is a pre-order traversal of the subtree of the AccessNode, or the complete syntactical tree of the AccessSuite.

TTCN/ASN.1 Derived Classes

The AccessVisitor class has one virtual member function for each TTCN and ASN.1 derived class in TTCN Access. Each of the member functions are declared

```
public:
virtual void Visit<class>( const <class> & );
```

Example 182 ----------

Visitor member method for an Identifier (excerpt from ITEXAccess-Visitor.hh):

```
class AccessVisitor
{
public:
    ...
  virtual void VisitIdentifier(const Identifier&);
  virtual void VisitVerdict(const Verdict&);
    ...
};
```

The base class implementation of this method calls the related Visit<child-class> function for all of the child objects to the current object.

If you need to change the behavior, for instance in order to generate some code or report from the TTCN Access Suite, just derive a new class from the AccessVisitor class and override the relevant method(s). Call the base class implementation of the method to traverse the children if needed.

Data Members

The AccessVisitor class contains no explicitly declared data members and is therefore stateless. It therefore makes program re-entrance possible, and several visitors may be active in the same tree/document at the same time if needed.

Using the AccessVisitor

The intended usage of the AccessVisitor class is by derivation. Derive one or several specialized classes for each of the purposes which you use TTCN Access. Override the relevant methods.

Use Case – Information Collection

It is simple to create a visitor class for collection of some kind of information from the test suite. The basic method for this is to have a handle or actual information in the visitor class.

Example 183 ----------

An information collecting visitor class:

```
#include <time.h>
#include <iostream.h>
#include <ITEXAccessClasses.hh>
#include <ITEXAccessVisitor.hh>

//
// A visitor which counts the number of testcases
// in a test suite.

class TestCounter : public AccessVisitor
{
public:
  TestCounter() : _count( 0 ) { }
  ~TestCounter() { cout << _count << endl; }
  void VisitTestCase(const TestCase&) { _count++; }
private:
  unsigned int _count;
};

// Small example usage, no real fault control...

int main( int argc, char ** argv )
{
  AccessSuite suite;
  if ( suite.open( argv[1] ) ) {
    TestCounter testCounter;
    testCounter.Visit( suite );
  }
  return 0;
}
```

Use Case – Code Generation

It is likewise simple to create a class for simple code generation. The following example is a class for generation of a c-style declaration which contains a list of all SimpleType names in a test suite.

Example 184 ----------

An information collecting visitor class:

```
#include <stdio.h>
#include <time.h>
#include <ITEXAccessClasses.hh>
#include <ITEXAccessVisitor.hh>

// A visitor which generate a c-style declaration
// of a null-terminated array of all Simple Type
// Declarations (identifiers) in a test suite.

class ListGenerator : public AccessVisitor
{
public:
  ListGenerator(FILE * f) : _file( f ) {
          printf( "const char *ids[]={\n");
        }
  void VisitSimpleTypeId(const SimpleTypeId& id) {
          printf( "  \"%s\",\n", (const char*)
                    id.simpleTypeIdentifier());
        }
  ~ListGenerator() {
          printf( "  NULL\n};\n");
        }
};

// Small example usage, no real fault control...

int main( int argc, char ** argv )
{
  AccessSuite suite;
  if ( suite.open( argv[1] ) ) {
    ListGenerator generator;
    generator.Visit( suite );
  }
  return 0;
}
```

The last two examples are not the most efficient implementations due to the fact that they traverse the whole tree even though we are only interested in small parts, and therefore we may want to improve the efficien-

cy. This may be done using any of the techniques described in "Optimizing Visitors" on page 978.

Advanced Use Case – Combined Visitors

More advanced designs may be possible by combining several visitors for performing more advanced tasks. You may for instance have visitors which are parameterized with other visitors for specialized tasks, where you still would want to maintain decent performance and yet not have trade-offs in clarity of the design.

Example 185 ––

A TTCN Interpreter would need a mechanism for building an internal representation of values. Values are always built by using the same structure, but you may wish to have several possible representations of atomic values. A visitor could be used to generate the value objects, where one visitor is used for building the overall structure, and another is used for building individual fields. There are at least two choices available in the design:

- To inherit the structure building class into the class which builds the atomic values
- To parameterize the structure building class with the atomic value handing class (which gives you a possibility to tune the behavior at runtime).

In this example, the value building class inherits from the AccessVisitor class and the Visit<xxx> functions are used to generate a new value. The class GciValueBuilder in the example, may visit any structured type and will on the Visit<class> function either create itself a data value, or if it is a structured type, create a dynamic array, and then invoke a new visitor for each of the fields, which results will later be assigned to each of the fields in the same array. The implementation is not present in the example. It is just outlined below.

The solution of using visitors for building the values, removes the switch statements, which otherwise would undoubtedly clobber an implementation which uses Direct Access as described in a previous section.

```
class GciValueBuilder : public AccessVisitor
{
  // basic structure for building/matching values
  virtual void VisitXxxx( const Xxxx & );
  ...
  // built value
  GciValue * _value;
};

class GciValueBinaryBuilder : public GciValueBuilder
{
  // suitable overrides for efficient binary value
  // handling
  ...
};

class GciValueBigNumBuilder : public GciValueBuilder
{
  // suitable ovverides for a 'bignum'
  // implementation
  ...
};
```

Optimizing Visitors

A visitor is a potentially inefficient way to find and process objects, since it in its unmodified version traverses the whole document, without regarding what parts of the document the inherited visitor is interested in. All optimizations are in the domain of limiting the subtree for which we are traversing. The following examples shows methods for avoiding unnecessary traversal and optimally, constant time access.

Example 186

Optimizing a visitor class to avoid traversal of all parts but the declarations part, may improve performance for suite traversal by over 100 times for some fairly representable suites (since most suites contain more and larger syntactical trees in the dynamic part than in all other parts, possibly with the exception of ASN.1 constraint declarations).

```
class DeclarationChecker : public AccessVisitor
{
 public:
  void VisitSuiteOverviewPart(
                const SuiteOverViewPart& ) { }
  void VisitConstraintsPart(
                const ConstraintsPart & ) { }
  void VisitDeclarationsPart (
                const DeclarationsPart & ) { }

  // Add the functions which actually do processing
  // below ...
};
```

Optimizing a visitor class to traverse only parts we are interested in, is also possible, by using one or a few levels of Direct Access instead of the default traversal.

Example 187 ----------

This example skips all traversal down to the Simple Type Definitions table, and only traverses those.

```
class SimpleTypeIdPrinter : public AccessVisitor
{
 public:
  void VisitASuite( const ASuite& );
  void VisitSimpleTypeId ( const SimpleTypeId & );
};

void
SimpleTypeIdPrinter::VisitASuite(const ASuite & s)
{
  VisitSimpleTypeDefs( s.declarationsPart().
                       definitions().
                       ts_TypeDefs().
                       simpleTypeDefs());
}

void
SimpleTypeIdPrinter::VisitSimpleTypeId( const
                                  SimpleTypeId & id )
{
  cout  << "Id: "
        << id.simpleTypeIdentifier()
        << endl;
}
```

Finally, you may combine several visitors into one visitor, thus avoiding multiple passes when processing a suite. This implies that you define several classes which are not truly visitors, and define a inherited class from AccessVisitor which calls methods in these classes.

Example 188 ----------

Two objects driven by a visitor, thus performing two passes on one traversal.

```
class IdOperation
{
 public:
  virtual void AtIdentifier(const Identifier&) = 0;
};

// A IdOperation

class IdCounter : public IdOperation
{
 public:
  IdCounter( ) : _count( 0 ) { }
  void AtIdentifier( const Identifier & id )
     { _count++; }
  ~IdCounter( ) { cout << _count << endl; }
 private:
  unsigned int _count;
};

// Another IdOperation

class IdPrinter : public IdOperation
{
  public:
   void AtIdentifier( const Identifier & id )
        { cout << id << endl; }
};

// A class which drives up to IdOpDriverMax
// IdOperations

const int IdOpDriverMax = 10;

class IdOpDriver : public AccessVisitor
{
 public:
  IdOpDriver() : _ops_used(0) { }
  void VisitIdentifier( const Identifier & id ) {
    for (unsigned int op = 0 ; op < _ops_used; ++op)
      _ops[op].AtIdentifier( id );
  }
  void AddIdOp( IdOperation * op )
        { _ops[_ops_used++] = op; }
```

```
 private:
  IdOperation  _ops[IdOpDriverMax];
  unsigned int _ops_used;
};

// Main routine which opens a suite and applies two
// IdOperations.

int main( int argc, char ** argv )
{
  AccessSuite suite;
  if ( suite.open( argv[1] ) ) {
    IdCounter  counter;
    IdPrinter  printer;
    IdOpDriver driver;
    driver.AddIdOp( &counter );
    driver.AddIdOp( &printer );
    driver.Visit( suite );
    suite.close( );
  }
  return 0;
}
```

Common Class Definitions

This part contains the declaration of the three common TTCN Access classes `AccessSuite`, `AccessNode` and `Astring`. For further information see TTCN Access include file `access.hh`.

AccessSuite

```
class AccessSuite
{
public:
  AccessSuite();
  ~AccessSuite();
  AccessSuite( const AccessSuite& orig );
  void operator=( const AccessSuite& orig );

  Boolean open( const char* suite_name );
  Boolean open( Suite* suite );
  Boolean close();

  const AccessNode root();
  const AccessNode find( const Identifier & id );
  const AccessNode find( const char* id );
 };
```

AccessNode

```
class AccessNode
{
public:
  AccessNode();
  AccessNode(NodeInfo nodeinfo);
  ~AccessNode();
  AccessNode(const AccessNode& Me);

  int operator==(const AccessNode& o) const;
  void operator=(const AccessNode& orig);

  Boolean is_equal(const AccessNode& o) const;
  Choices::Choice choice() const;
  Boolean ok() const;

};
```

Astring

```
class Astring
{
public:
  Astring();
  Astring(const char* s);
  // end should point to the char after the last
char
  Astring(const char* begin, const char* end);
  Astring(Field* field, PT* pt);
  Astring(const Astring& orig);

  ~Astring();

  //operators
  Astring* operator->();
  const Astring* operator->() const;
  operator const char*() const;
  char& operator[](unsigned i) ;
  char operator[](unsigned i) const ;

  void operator=(const Astring&);
  void operator=(const String&);
  void operator=(const char*);
  void operator=(const char);

  int operator==(const Astring& s) const;
  int operator!=(const Astring& s) const;
  int operator==(const char* cs) const;
  int operator!=(const char* cs) const;
};
```

Getting Started with TTCN Access

Setting Up the TTCN Access Environment

When building applications using TTCN Access, the compiler will need to find a few files, the `ITEXAccessClasses.hh` include file and the `libaccess.a` library file. Normally these files are found in the `.../itex/include/CC` and `.../itex/lib/CC` directory respectively. The include file must be included in every TTCN Access application and the library `libaccess.a` must be used when linking.

> **Note:**
>
> For further information, contact your system administrator.

TTCN Access operates on the TTCN Suite data bases. These data bases must have passed analysis and be saved before using TTCN Access. If the data base contains a TTCN test suite that is not analyzed, the TTCN Access application can not reach into the fields of the tables. The default behavior of TTCN Access is to simply skip those fields that are not analyzed. De-referencing a particular field in an non-analyzed data base, will result in undefined behavior.

TTCN test suite data bases are managed by the AccessSuite object, which has member functions for opening and closing the TTCN Suite data bases and also for starting traversing. From an AccessSuite object it is also possible to access tables in a random manner via the symbol table manager.

Using Example Applications

TTCN Access is delivered with some simple example applications. To compile the examples, the installation directory, where the `ITEXAccessClasses.hh` and `libaccess.a` files reside, must be known to the makefiles. Do this by setting the environment variable ACCESS or by explicitly filling in the local variable ACCESS in every makefile or using make with the syntax `make ACCESS=$telelogic/itex/access`.

Starting an TTCN Access Application

You have to retrieve the TTCN Access license when you start the TTCN Suite. To do this, start the TTCN Suite from the command line with the switch `-access`.

Then you can execute TTCN Access applications from the command line or from the window manager.

You can also select *Start Application* in the *Access* menu in the Browser. This will open a dialog in which you may change settings and start the TTCN Access application.

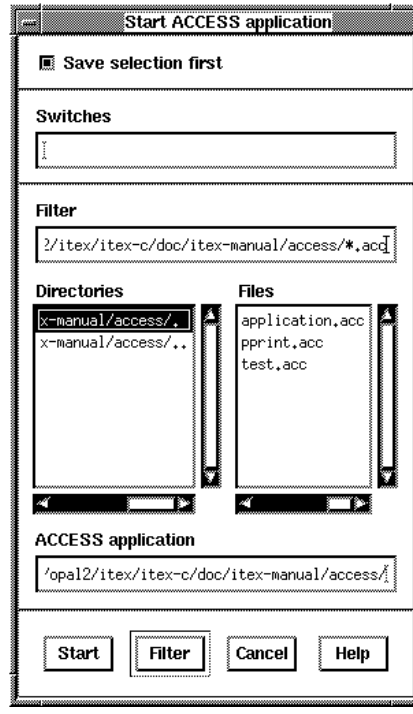


Figure 181: The Start Application dialog

Save selection first

Select this to save the current selection in the Browser before executing the TTCN Access application.

Switches

Switches may be passed to the chosen TTCN Access application. The switches are written as free text. As the last argument is the name of the the TTCN Suite database passed. This name is always passed to the TTCN Access application.

> **Note:**
>
> Observe that it is the name of the (working) *database* file and not the name of the test suite that is passed to the TTCN Access application.

Filter

Sets the filter for the files that will be displayed in the *Suites* list. There are no predefined naming conventions for TTCN Access applications.

For example the name filter `*.acc` will cause only those files whose names end with `.acc` to be displayed.

Access application

Displays the selected application.

Chapter

22

Creating a TTCN Access Application

This chapter is intended to be an introduction to how to create a TTCN Access application, and it is expected to be read sequentially. As TTCN Access is a C++ application programmers interface, this chapter will contain several TTCN Access applications written in C++. The chapter includes general TTCN Access application concepts as well as specific examples.

For reference, the test suite that is used as input to the various TTCN Access applications, is included in TTCN-GR form at the end of this chapter together with the source code for each example.

TTCN Access

TTCN Access is a C++ application programmers interface. It is a platform for writing applications related to TTCN test suites such as:

- Executable test suites
- Interpreters
- Encoders and decoders
- Reporters

TTCN Access is meant to be used by executable test suite developers, translator developers as well as test result analyzers.

In order to make this chapter as complete as possible and thereby cover as many different TTCN Access application domains as we can, this chapter contains several different examples, each described in its own section. However, before we start with the examples, we will explain some general concepts that apply to every TTCN Access application.

Note:

This chapter is an TTCN Access primer and is not intended to provide a chapter in C++ programming.

General Concepts

The TTCN Access Application

A TTCN Access application is typically an application that produces some kind of output during the traversing of an Access tree. The Access tree, tree for short, is a tree oriented representation of a test suite. This tree is stored in the database `<TestSuite>.itex` where `<TestSuite>` is the user given name of the database.

The first section introduces the basic methods. These include the main function and how to initialize a test suite in a TTCN Access application. The second section describes how the traversing is done.

Basic Methods

When executing a TTCN Access application there are some basic methods that have to be applied before and after traversing the tree. First of all, to be able to get all the definitions needed, the header files `ITEX-`

`AccessClasses.hh` and `ITEXAccessVisitor.hh` have to be included. These files must be included in all TTCN Access applications as shown below:

```
#include <ITEXAccessClasses.hh>
#include <ITEXAccessVisitor.hh>
```

Once this is done, we need to create an instance of the *AccessSuite* class. This can be done as follows:

```
AccessSuite suite;
```

The *AccessSuite* class can be seen as a handle to the Access tree and the symbol table. It will be used for opening and closing the database and for finding specific, named Access objects stored in the symbol table.

To open a database for reading, the following method has to be called:

```
Boolean open_ok = suite.open( arg );
```

where `arg` is a string containing the name of the database that shall be opened. This method must always be applied to the database before traversing the Access tree. The method returns a boolean value stating whether the database could be opened or not. To close a database the following method must be called:

```
Boolean close_ok = suite.close();
```

This method requires no database name to be given as the database that will be closed is associated with `suite`. The method returns a boolean value stating whether the database could be closed or not. Below is a generic TTCN Access main function where `argv[1]` contains the name of the database:

```
#include <ITEXAccessClasses.hh>
#include <ITEXAccessVisitor.hh>

int main( int argc, char **argv )
{
  AccessSuite suite;

  if ( suite.open( argv[ 1 ] ) )       // OK open database?
    {
      // Write your Access application here
      // Typically start the traverser

     if ( suite.close( ) )            // OK close database?
       return 1;
     else                             // Error close database!
       return 0;
     }
   else                               //Error open database!
       return -1;
}
```

Traversing the Access Tree

The traversing of a test suite is the “engine” that actually drives the TTCN Access application to execute.

The previous section was only concerned with opening and closing of the database. This section will describe and explain the different methods that TTCN Access provides for starting the actual TTCN Access application, i.e. starting the traversing from different nodes in the Access tree.

To be able to allow the user to traverse the tree in a simple way, TTCN Access provides a visitor class that can be customized in a flexible way by the user. The class is called *AccessVisitor* and provides different ways of traversing.

In all the examples discussed in the following sub sections, we assume that we have a global instance of the class *AccessSuite* called *suite, a global instance of the class AccessVisitor called visitor* and that a test suite has been opened as described in the previous section.

Traversing from the Root of the Tree

As TTCN Access can be seen as a huge tree, the basic method for traversing will be to start traversing from the root. This will result in a traversing that will traverse the full Access tree starting from top. This is performed by calling the method:

```
Boolean root_ok = visitor.Visit( suite );
```

It is an error to try to traverse a database that has not been previously opened.

Traversing by Reference

In some cases it is desirable to start traversing from a specific table, i.e. a test case or a type definition. TTCN Access provides means for this in terms of using its symbol table.

The symbol table handles global entities, that is elements that must be globally unique within a test suite according to the TTCN standard (9646-3). Such elements are all variable names, all type names, all test case names, etc.

To start traversing the Access tree from a specific element, TTCN Access provides a simple solution divided into four steps:

1. The first step is to get a handle to the symbol table:

   ```
   AccessNode node = suite.find( name );
   ```

   This method will return a handle to the symbol table.

2. The second step is to verify that this handle holds an element present in the symbol table. This is performed by applying the following method:

   ```
   Boolean symbol_table_ok = node.ok();
   ```

3. Assuming that the argument `name` was a global name in the test suite, we still do not know what kind of element it is. It could for example be a TTCN PDU type definition, a test case or a test step. The handle to the symbol table provides a method for finding the Access type of its corresponding element. The third step will be to ask the symbol table what type its corresponding element has:

   ```
   Choices::Choice type = node.choice();
   ```

   This method will return an enumerated value of type `Choices::Choice` revealing the type of the element. As there are several different Access types, this type check is best solved in a switch statements. Assuming that the name can be either a test case or a TTCN PDU type definition, the following switch statement will detect that.

   ```
   switch( node.choice( ) )
   {
     case Choices::c_TTCN_PDU_TypeDef:
           //
           // do something
           //
           break;
     case Choices::c_TestCase:
           //
           // do something else
           //
           break;
     default:
           //
           // do something default
           //
           break;
   }
   ```

4. The fourth step is to call the corresponding traverser function.

   Below is a complete example that will traverse a test suite starting from a specific element. The example assumes that the function `main( )` is called with two arguments where the first is the name of the database and the second is the name of the element that the traversing shall start from. This application can only handle TTCN PDU type definitions and test cases.

```
int main( int argc, char **argv )
{
  AccessSuite suite;
  AccessVisitor visitor;

  if ( suite.open( argv[ 1 ] ) )        // OK open database?
    {
      //Get the handle

      AccessNode node = suite.find( arg[ 2 ] );

      if ( node.ok( ) )                 // Handle ok?
        visitor.Visit( node );
      else                              // Error in symbol table
        return -2

      if ( suite.close( ) )             // OK close database?
        return 1;
      else                              // Error close database!
        return 0;
     }
   else                                 // Error open database!
     return -1;
}
```

Examples

This section provides all the examples in this chapter. It starts with a simple TTCN Access application and continues from there. Every example is discussed in its own subsection. In each subsection, only the interesting parts are discussed. A complete solution to every example can be found in "Solutions to the Examples" on page 1000.

Every TTCN Access application can be seen as an application providing a solution to a problem. These problems are related to TTCN test suites in general, such as reporters or style checkers, encoders and decoders. Therefore, every example in this chapter will be preceded with a small discussion of the actual problem that the corresponding TTCN Access application shall solve. Of course, the problems discussed in this chapter will mainly be of a fictive nature, but they will reflect the simplicity of TTCN Access and the different implementation techniques and methods that TTCN Access provides.

In all examples, we assume that there exist a global instance of the class *AccessSuite* called *suite*, a global instance of the class *AccessVisitor* called *visitor* and that a test suite has been opened.

The Identifiers

Objective

In every test suite there exists a set of identifiers. The objective of this example is to print out every identifier present in the test suite.

Solution 1

Every identifier present in an Access tree will be represented as an Access node of type Identifier. The solution will therefore only include one user defined method in the visitor class instance that, when entering the node Identifier, prints the content of that class. In this case it is the name of the identifier. The method is provided below.

```
void Trav::VisitIdentifier( const Identifier& Me )
{
  printf( "Identifier %s\n", (const char*) Me );
}
```

This method will be called from TTCN Access every time the traverser enters the Identifier node. The name `Identifier` is the name of the

Access node and the prefix tag `Visit` means that this is a visiting method.

As the task of a TTCN Access application is to traverse a test suite according to the traverser and call possibly user defined functions or methods, we need to create a traverser that will use the default traversing, but have the desired small change. This is done by sub classing as follows:

```
class Trav : public AccessVisitor
{
public:
  void VisitIdentifier( const Identifier& id );
};
```

As identifiers are present everywhere in the Access tree, no change to the traversing order needs to be done. The default traverser will be used and the traversing will start from root.

The full solution is found in "Solution 1" on page 1001.

Context

Objective

When executing the previous example, all identifiers in the given test suite will be printed. By looking at the output one might think that there should not be so many identifiers in the test suite, but the way in which the grammars are specified (both TTCN and ASN.1), nearly every grammar rule ends in a identifier. This example will explain the importance of calling the user defined function at their correct place, or to be more specific, in its right context.

Let us assume, with the previous example in mind, that we want to distinguish between TTCN PDU type definition identifiers and TTCN PDU constraint identifiers (the rest is of no concern). This can be achieved in several different ways. Below are two different solutions to this problem.

Solution 2

This first solution, looking at each identifiers type, will be implemented in the visiting method used in the previous example. However, it has to be extended to look at each identifier type and only print the appropriate ones. The updated method is found below.

```
void Trav::VisitIdentifier( const Identifier& Me )
{
  switch( Me.choice( ) )
    {
      case Choices::c_TTCN_PDU:
        printf( "Identifier '%s' is a TTCN PDU type\n",
                (const char*) Me );
        break;

      case Choices::c_TTCN_PDU_Cons:
        printf( "Identifier '%s' is a TTCN PDU constraint\n",
                (const char*) Me );
        break;

      default:
        break;
    }
}
```

Solution 3

The second solution will replace the used method with two new ones. The first method will print out every TTCN PDU type definition identifier. This will be done in the method `VisitTTCN_PDU_TypeDef`. The second, `VisitTTCN_PDU_Constraint`, will print out every TTCN PDU constraint identifier. In each method the Identifier Access node is reached by explicit addressing, that is, explicitly naming the path to the Access node Identifier. Their corresponding definitions are supplied below.

```
void Trav::VisitTTCN_PDU_TypeDef( const TTCN_PDU_TypeDef& Me )
{
  printf( "Identifier '%s' is a TTCN PDU type\n",
           (const char*) Me.pdu_Id( ).pdu_IdAndFullId( )
           .pdu_Identifier( ) );
}

void Trav::VisitTTCN_PDU_Constraint( const
TTCN_PDU_Constraint& Me)
{
  printf( "Identifier '%s' is a TTCN PDU constraint\n",
          (const char*) Me.consId( ).consIdAndParList( ).
          .constraintIdentifier( ) );
}
```

The definition of the visitor object is done as follows:

```
class Trav : public AccessVisitor
{
public:
  void VisitTTCN_PDU_TypeDef( const
TTCN_PDU_TypeDef& );
  void VisitTTCN_PDU_Constraint( const
TTCN_PDU_Constraint& );
};
```

Comments

When executing the two examples above with the test suite in the end of this chapter as input, one will find that the output is not equivalent. The first solution will have some more output lines that the second one. By adding the following line to each method, one will detect in which table the identifier was found.

```
printf( "In table '%s': ", (const char*) Me.table_name() );
```

One will find that the extra lines of output in the first solution are produced from the Constraints Part and the Dynamic Part. This is true due to the fact that the type identifiers are being used in each constraint and the constraint identifiers are being used in the Constraint Reference column in each test case and test step table.

The full solutions are found in "Solution 2" on page 1003 and "Solution 3" on page 1005.

Translating TTCN

A very common and useful TTCN Access application is to translate a test suite to another target language. Examples are C, C++, Pascal etc. By doing this translation we can thereby create an executable test suite. To achieve this, one needs a TTCN translator that translates a TTCN test suite to the specific source code. This example will not be a full translator, but will rather show how such a translator can be written by using TTCN Access.

Objective

In this example, we will translate TTCN PDU type definitions to C where each PDU will be a unique C type. To achieve this, we need to have some kind of mapping between TTCN and C. One way to do this is to do a case study. First we will look at a TTCN PDU type definition in the table format and from that table extract how the corresponding C representation shall be. Then, a generic representation of all TTCN PDUs in C will be defined. This generic representation will then be implemented in TTCN Access, and thereby a generic TTCN PDU type definition to C translator will be implemented.

Solution 4

Let us have a look at a TTCN PDU type definition in table format. The table below, the TTCN PDU type definition SEND, is taken from the test suite attached to the end of this chapter.

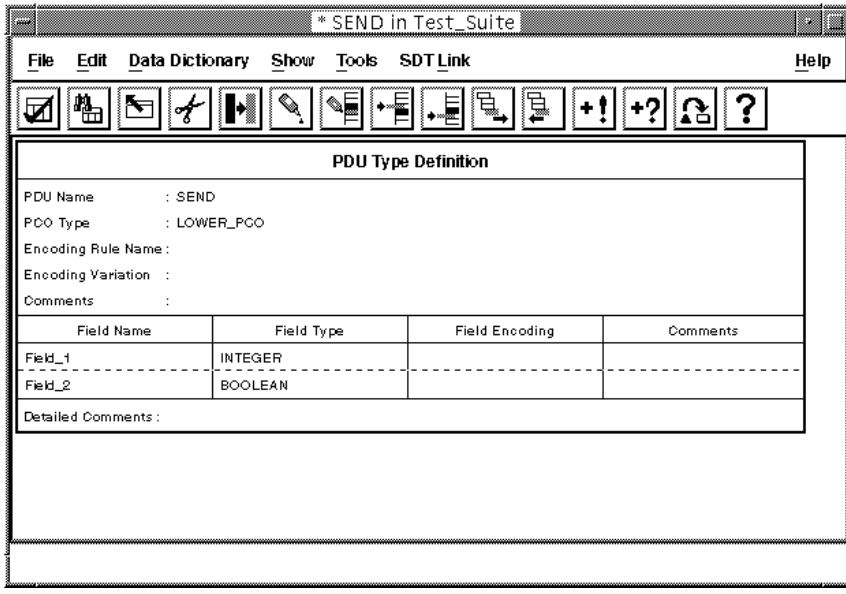


Figure 182: TTCN PDU Type Definition

A corresponding C representation of the above type could look something like:

```
typedef struct {
 INTEGER Field_1;
 BOOLEAN Field_2;
} SEND;
```

That is, a structured type holding the same number of elements as the corresponding TTCN type definition. In this example, no consideration is taken to the PCO type. This is only done for simplicity and can of course be included in the type as well.

As one can see in this example there are a lot similarities between the TTCN definition and the C representation in terms of names of elements

and their corresponding type. This helps us define a generic C representation of a TTCN PDU type definition as below:

```
typedef struct {
 Element1_Type Element1_Name;
 Element2_Type Element2_Name;
 ...
 ElementN_Type ElementN_Name;
} PDU_TypeName;
```

The type and name of each element is taken from the corresponding names in the TTCN type definition and the type defined is named as in TTCN.

Now, the next step is to implement this generic type in TTCN Access! To do this, we divide the generic C type into three distinct elements; the header, the body and the footer.

The Header and Footer

The visitor method that traverses the TTCN PDU type definitions, will be the one responsible for generating the header and footer of the generated C structure definition. The method may look something like this:

```
void Trav::VisitTTCN_PDU_TypeDef( const TTCN_PDU_TypeDef& Me )
{
  // Generate the header part

  cout << Me.pdu_Id( ) << "_encode( const ";
  cout << Me.pdu_Id( ) << "& me, char* enc_buf )";
  cout << "\n{" << endl;

  // Traverse fields using default traverser

  AccessVisitor::VisitPDU_FieldDcls(
Me.pdu_FieldDcls( ) );

  // Generate the footer part

  cout << "}" << endl;
}
```

This method will be called every time the traverser enters the Access node TTCN_PDU_TypeDef.

The Body

The body of the type definition, is a collection of rows where each row is translated identically. A row can be translated by defining the following method:

```
void Trav::VisitPDU_FieldDcl( const PDU_FieldDcl& Me )
{
  printf( "  %s %s;\n",
          (const char*) get_type_str( Me.pdu_FieldType( ),
          (const char*) get_id( Me.pdu_FieldId( ) );
}
```

where `get_type_str` and `get_id` are two functions returning the type and the name of the element respectively. This method shall be called every time the traverser enters the Access node PDU_FieldDcl.

The full solution is found in "Solution 4" on page 1006.

Summary

TTCN Access provides a platform for generating applications in the domain of TTCN, conformance testing and related applications such as translators, interpreters, analyzers, reporters, etc. TTCN Access is a C++ representation of the related grammars that spans the language TTCN. We hope that this chapter has revealed the basis of TTCN Access, the methods it supplies and how easy it is to use. All examples in this chapter have full solutions in "Solutions to the Examples" on page 1000.

Solutions to the Examples

This section contains fully implemented solutions to each example given in the "Examples" on page 993. It will also contain the expected output when executing each TTCN Access application with the example test suite as input.

Makefile

This generic Makefile has been used for compilation and linking:

```
#-------------------------------------------------------------
# Name: Makefile
#
# Destcription:
# Generic Makefile for Access applications
# Note that the makefile requires the variable
# itexaccess to be set.
#-------------------------------------------------------------

ACCESS-INCL = $(itexaccess)/include
ACCESS-LIB = $(itexaccess)/lib

TARGET = the name of the target
OBJS = $(TARGET).o

#-----------------------------------------------------------

$(TARGET) :$(OBJS)
        $(CCC) -L$(ACCESS-LIB) -Bstatic -o $@ $(OBJS) -laccess
        strip $@

#-----------------------------------------------------------

clean :
        /bin/rm $(TARGET) $(OBJS)

#-------------------------------------------------------------

.SUFFIXES: .cc .o

.cc.o :
        $(CCC) -I$(ACCESS-INCL) -c -o $@ $<
```

Solution 1

Code

```
#include <ITEXAccessClasses.hh>
#include <ITEXAccessVisitor.hh>

class Trav : public AccessVisitor
{
public:
  void VisitIdentifier( const Identifier& id );
};


void Trav::VisitIdentifier( const Identifier& Me )
{
  printf( "Identifier %s\n", (const char*) Me );
}


int main( int /* argc */, char **argv )
{
  AccessSuite suite;

  if( suite.open( argv[ 1 ] ) )   // OK open database?
   {
      Trav visitor;

      visitor.Visit( suite );

      if( suite.close( ) )       // OK close database?
        return 1;
      else                       // Error close database!
        return 0;
   }

  else                           // Error open database!
    return -1;
}
```

Output

```
> ./print-ids ../Example_Suite_A.itex
Identifier Example_Suite_A
Identifier Example_Suite_A
Identifier L
Identifier LOWER_PCO
Identifier SEND
Identifier LOWER_PCO
Identifier Field_1
Identifier Field_2
Identifier RECEIVE
Identifier LOWER_PCO
Identifier Field_1
Identifier Field_2
Identifier S1
Identifier SEND
Identifier Field_1
Identifier Field_2
Identifier R1
Identifier RECEIVE
Identifier Field_1
Identifier Field_2
Identifier TEST_CASE_1
Identifier L
Identifier SEND
Identifier S1
Identifier TEST_STEP_1
Identifier TEST_STEP_1
Identifier L
Identifier RECEIVE
Identifier R1
Identifier L
>
```

Solution 2

Code

```
#include <ITEXAccessClasses.hh>
#include <ITEXAccessVisitor.hh>

class Trav : public AccessVisitor
{
public:
  void VisitIdentifier( const Identifier& Me );
};


int main( int /* argc */, char **argv )
{
  AccessSuite suite;

  if( suite.open( argv[ 1 ] ) )   // OK open database?
   {
      Trav visitor;

      visitor.Visit( suite );

      if( suite.close( ) )       // OK close database?
        return 1;
      else                       // Error close database!
        return 0;
   }

  else                           // Error open database!
    return -1;
}


void Trav::VisitIdentifier( const Identifier& Me )
{
  switch( Me.choice( ) )
    {
      case Choices::c_TTCN_PDU:
        printf( "Identifier '%s' is a TTCN PDU type\n",
                (const char*) Me );
        break;

      case Choices::c_TTCN_PDU_Cons:
        printf( "Identifier '%s' is a TTCN PDU constraint\n",
                (const char*) Me );
        break;

      default:
        break;
    }
}
```

Output

```
> ./print-ids ../Example_Suite_A.itex

Identifier 'SEND' is a TTCN PDU type
Identifier 'RECEIVE' is a TTCN PDU type
Identifier 'S1' is a TTCN PDU constraint
Identifier 'SEND' is a TTCN PDU type
Identifier 'R1' is a TTCN PDU constraint
Identifier 'RECEIVE' is a TTCN PDU type
Identifier 'SEND' is a TTCN PDU type
Identifier 'S1' is a TTCN PDU constraint
Identifier 'RECEIVE' is a TTCN PDU type
Identifier 'R1' is a TTCN PDU constraint
>
```

Solution 3

Code

```
#include <access.hh>

void id_pre_TTCN_PDU_TypeDef(const TTCN_PDU_TypeDef& Me);
void id_pre_TTCN_PDU_Constraint(const TTCN_PDU_Constraint&
Me);

int main( int /* argc */, char **argv )
{
  AccessSuite suite;

  if( suite->open( argv[ 1 ] ) )   // OK open database?
    {
      // Setup the id writing functions
      pre_TTCN_PDU_TypeDef = id_pre_TTCN_PDU_TypeDef;
      pre_TTCN_PDU_Constraint = id_pre_TTCN_PDU_Constraint;

      // Start traversing from root.
      if( !suite->trav_root() )
        return 1;

      if( suite->close() )         // OK close database?
        return 0;
       else                        // Error close database!
        return 2;
    }

  else                             // Error open database!
   return -1;
}


void id_pre_TTCN_PDU_TypeDef(const TTCN_PDU_TypeDef& Me)
{
  printf( "Identifier '%s' is a TTCN PDU type\n",
           (const char*) Me->pdu_Id()->pdu_IdAndFullId()
           ->pdu_Identifier() );
}


void id_pre_TTCN_PDU_Constraint(const TTCN_PDU_Constraint& Me)
{
  printf( "Identifier '%s' is a TTCN PDU constraint\n",
          (const char*) Me->consId()->consIdAndParList()
          ->constraintIdentifier() );
}
```

Output

```
> print-ids ../Example_Suite_A.itex

Identifier 'SEND' is a TTCN PDU type
Identifier 'RECEIVE' is a TTCN PDU type
Identifier 'S1' is a TTCN PDU constraint
Identifier 'R1' is a TTCN PDU constraint
>
```

Solution 4

Code

```
#include <ITEXAccessClasses.hh>
#include <ITEXAccessVisitor.hh>

class Trav : public AccessVisitor
{
public:
  void VisitTTCN_PDU_TypeDef( const TTCN_PDU_TypeDef& Me );
  void VisitPDU_FieldDcl( const PDU_FieldDcl& Me );
};

Astring get_type_str(const AccessNode& n);
Astring get_id(const AccessNode& n);

int main( int /* argc */, char **argv )
{
  AccessSuite suite;

  if( suite.open( argv[ 1 ] ) )   // OK open database?
    {
      Trav visitor;

      visitor.Visit( suite );

      if( suite.close() )         // OK close database?
        return 0;
      else                        // Error close database!
        return 2;
    }

  else                            // Error open database!
    return -1;
}


void Trav::VisitTTCN_PDU_TypeDef( const TTCN_PDU_TypeDef& Me )
{
  // Generate the header part

  cout << Me.pdu_Id( ) << "_encode( const ";
  cout << Me.pdu_Id( ) << "& me, char* enc_buf )";
  cout << "\n{" << endl;

  // Traverse fields using default traverser

  AccessVisitor::VisitPDU_FieldDcls( Me.pdu_FieldDcls( ) );

  // Generate the footer part

  cout << "}" << endl;
}


void Trav::VisitPDU_FieldDcl( const PDU_FieldDcl& Me )
{
  printf( "  %s %s;\n",
          (const char*) get_type_str( Me.pdu_FieldType( ),
          (const char*) get_id( Me.pdu_FieldId( ) );
}
```

```
// Help functions

Astring get_type_str(const AccessNode& n)
{
  switch ( n.choice( ) )
    {
    case Choices::c_PDU_FieldType:
      return n.PDU_FieldType().content();

    default:
      fprintf( stderr, "Bad case in get_type %d\n",
               (int) n.choice( ) );
      abort();
    }
}


Astring get_id(const AccessNode& n)
{
  switch ( n.choice( ) )
    {
      case Choices::c_PDU_FieldId:
           if ( n.PDU_FieldId( ).pdu_FieldIdOrMacro( )
                .choice( )
                == Choices::c_MacroSymbol )
           return n.PDU_FieldId( ).pdu_FieldIdOrMacro( )
                   .macroSymbol( );
        else
           return n.PDU_FieldId( ).pdu_FieldIdOrMacro( )
                   .pdu_FieldIdAndFullId( )
                   .pdu_FieldIdentifier( );

      case Choices::c_PDU_Id:
        return n.PDU_Id( ).pdu_IdAndFullId( )
               .pdu_Identifier( );

      default:
        fprintf( stderr, "Bad case in get_id %d\n",
                 (int) n.choice( ) );
        abort( );
    }
}
```

Output

```
> ./pdu2c ../Example_Suite_A.itex

typedef struct {
  INTEGER Field_1;
  BOOLEAN Field_2;
} SEND;

typedef struct {
  INTEGER Field_1;
  BOOLEAN Field_2;
} RECEIVE;
>
```

Chapter 22 Creating a TTCN Access Application

Chapter

23

The TTCN Access Class Reference Manual

This chapter contains the class definitions in TTCN Access.

The definitions are grouped into the following areas:

Static and Table Nodes

This section contains all static nodes and table nodes (that is those visible in the Browser) and also the nodes representing rows inside tables. Common for these nodes are that they are always present, there may be any number of them, but they are generally well defined and cannot just be missing.

AliasDef ::= SEQUENCE {

aliasId	AliasId	FIELD	--See page 1042
expandedId	ExpandedId	FIELD	--See page 1059
comment	Comment	FIELD	--See page 1097

}

Referenced from: AliasDefList (p. 1010)

AliasDefList ::= SEQUENCE OF {

aliasDef	AliasDef		--See page 1010

}

Referenced from: AliasDefs (p. 1010)

AliasDefs ::= SEQUENCE {

aliasDefList	AliasDefList		--See page 1010
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: ComplexDefinitions (p. 1019)

ASN1_ASP_Constraint ::= SEQUENCE {

consId	ConsId	FIELD	--See page 1050
asp_Id	ASP_Id	FIELD	--See page 1044
derivPath	DerivPath	FIELD	--See page 1056
comment	Comment	FIELD	--See page 1097
asn1_ConsValue	ASN1_ConsValue	FIELD	--See page 1043
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: ASN1_ASP_Constraints (p. 1010)

ASN1_ASP_Constraints ::= SEQUENCE OF {

asn1_ASP_Constraint	ASN1_ASP_Constraint		--See page 1010

}

Referenced from: ASP_Constraints (p. 1015)

ASN1_ASP_TypeDef ::= SEQUENCE {

asp_Id	ASP_Id	FIELD	--See page 1044
pco_Type	PCO_Type	FIELD	--See page 1071
comment	Comment	FIELD	--See page 1097
asn1_TypeDefinition	ASN1_TypeDefinition	FIELD	--See page 1043
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: ASN1_ASP_TypeDefs (p. 1011)

ASN1_ASP_TypeDefByRef ::= SEQUENCE {

asp_Id	ASP_Id	FIELD	--See page 1044
pco_Type	PCO_Type	FIELD	--See page 1071
asn1_TypeReference	ASN1_TypeReference	FIELD	--See page 1096
asn1_ModuleId	ASN1_ModuleId	FIELD	--See page 1096
comment	Comment	FIELD	--See page 1097
asn1_TypeDefinition	ASN1_TypeDefinition	FIELD	--See page 1043

}

Referenced from: ASN1_ASP_TypeDefByRefList (p. 1011)

ASN1_ASP_TypeDefByRefList ::= SEQUENCE OF {

asn1_ASP_TypeDefByRef	ASN1_ASP_TypeDefByRef	--See page 1011

}

Referenced from: ASN1_ASP_TypeDefsByRef (p. 1011)

ASN1_ASP_TypeDefs ::= SEQUENCE OF {

asn1_ASP_TypeDef	ASN1_ASP_TypeDef	--See page 1011

}

Referenced from: ASP_TypeDefs (p. 1016)

ASN1_ASP_TypeDefsByRef ::= SEQUENCE {

asn1_ASP_TypeDefByRefList	ASN1_ASP_TypeDefByRefList		--See page 1011
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: ASP_TypeDefs (p. 1016)

ASN1_CM_Constraint ::= SEQUENCE {

consId	ConsId	FIELD	--See page 1050
cm_Id	CM_Id	FIELD	--See page 1049
derivPath	DerivPath	FIELD	--See page 1056
comment	Comment	FIELD	--See page 1097
asn1_ConsValue	ASN1_ConsValue	FIELD	--See page 1043
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: ASN1_CM_Constraints (p. 1012)

ASN1_CM_Constraints ::= SEQUENCE OF {

asn1_CM_Constraint	ASN1_CM_Constraint	--See page 1012

}

Referenced from: CM_Constraints (p. 1017)

ASN1_CM_TypeDef ::= SEQUENCE {

cm_Id	CM_Id	FIELD	--See page 1049
comment	Comment	FIELD	--See page 1097
asn1_TypeDefinition	ASN1_TypeDefinition	FIELD	--See page 1043
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: ASN1_CM_TypeDefs (p. 1012)

ASN1_CM_TypeDefs ::= SEQUENCE OF {

asn1_CM_TypeDef	ASN1_CM_TypeDef	--See page 1012

}

Referenced from: CM_TypeDefs (p. 1018)

ASN1_PDU_Constraint ::= SEQUENCE {

consId	ConsId	FIELD	--See page 1050
pdu_Id	PDU_Id	FIELD	--See page 1072
derivPath	DerivPath	FIELD	--See page 1056
encRuleId	EncRuleId	FIELD	--See page 1058
encVariationId	EncVariationId	FIELD	--See page 1058
comment	Comment	FIELD	--See page 1097
asn1_ConsValue	ASN1_ConsValue	FIELD	--See page 1043
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: ASN1_PDU_Constraints (p. 1013)

ASN1_PDU_Constraints ::= SEQUENCE OF {

asn1_PDU_Constraint	ASN1_PDU_Constraint	--See page 1012

}

Referenced from: PDU_Constraints (p. 1026)

ASN1_PDU_TypeDef ::= SEQUENCE {

pdu_Id	PDU_Id	FIELD	--See page 1072
pco_Type	PCO_Type	FIELD	--See page 1071
pdu_EncodingId	PDU_EncodingId	FIELD	--See page 1071
encVariationId	EncVariationId	FIELD	--See page 1058
comment	Comment	FIELD	--See page 1097
asn1_TypeDefinition	ASN1_TypeDefinition	FIELD	--See page 1043
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: ASN1_PDU_TypeDefs (p. 1013)

ASN1_PDU_TypeDefByRef ::= SEQUENCE {

pdu_Id	PDU_Id	FIELD	--See page 1072
pco_Type	PCO_Type	FIELD	--See page 1071
asn1_TypeReference	ASN1_TypeReference	FIELD	--See page 1096
asn1_ModuleId	ASN1_ModuleId	FIELD	--See page 1096
pdu_EncodingId	PDU_EncodingId	FIELD	--See page 1071
encVariationId	EncVariationId	FIELD	--See page 1058
comment	Comment	FIELD	--See page 1097
asn1_TypeDefinition	ASN1_TypeDefinition	FIELD	--See page 1043

}

Referenced from: ASN1_PDU_TypeDefByRefList (p. 1013)

ASN1_PDU_TypeDefByRefList ::= SEQUENCE OF {

asn1_PDU_TypeDefByRef	ASN1_PDU_TypeDefByRef	--See page 1013

}

Referenced from: ASN1_PDU_TypeDefsByRef (p. 1014)

ASN1_PDU_TypeDefs ::= SEQUENCE OF {

asn1_PDU_TypeDef	ASN1_PDU_TypeDef	--See page 1013

}

Referenced from: PDU_TypeDefs (p. 1026)

ASN1_PDU_TypeDefsByRef ::= SEQUENCE {

asn1_PDU_TypeDefByRefList	ASN1_PDU_TypeDefByRefList		--See page 1013
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: PDU_TypeDefs (p. 1026)

ASN1_TypeConstraint ::= SEQUENCE {

consId	ConsId	FIELD	--See page 1050
asn1_TypeId	ASN1_TypeId	FIELD	--See page 1044
derivPath	DerivPath	FIELD	--See page 1056
encVariationId	EncVariationId	FIELD	--See page 1058
comment	Comment	FIELD	--See page 1097
asn1_ConsValue	ASN1_ConsValue	FIELD	--See page 1043
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: ASN1_TypeConstraints (p. 1014)

ASN1_TypeConstraints ::= SEQUENCE OF {

asn1_TypeConstraint	ASN1_TypeConstraint		--See page 1014

}

Referenced from: TS_TypeConstraints (p. 1037)

ASN1_TypeDef ::= SEQUENCE {

asn1_TypeId	ASN1_TypeId	FIELD	--See page 1044
encVariationId	EncVariationId	FIELD	--See page 1058
comment	Comment	FIELD	--See page 1097
asn1_TypeDefinition	ASN1_TypeDefinition	FIELD	--See page 1043
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: ASN1_TypeDefs (p. 1014)

ASN1_TypeDefs ::= SEQUENCE OF {

asn1_TypeDef	ASN1_TypeDef		--See page 1014

}

Referenced from: TS_TypeDefs (p. 1037)

ASN1_TypeRef ::= SEQUENCE {

asn1_TypeId	ASN1_TypeId	FIELD	--See page 1044
asn1_TypeReference	ASN1_TypeReference	FIELD	--See page 1096
asn1_ModuleId	ASN1_ModuleId	FIELD	--See page 1096
encVariationId	EncVariationId	FIELD	--See page 1058
comment	Comment	FIELD	--See page 1097
asn1_TypeDefinition	ASN1_TypeDefinition	FIELD	--See page 1043

}

Referenced from: ASN1_TypeRefList (p. 1015)

ASN1_TypeRefList ::= SEQUENCE OF {

asn1_TypeRef	ASN1_TypeRef		--See page 1015

}

Referenced from: ASN1_TypeRefs (p. 1015)

ASN1_TypeRefs ::= SEQUENCE {

asn1_TypeRefList	ASN1_TypeRefList		--See page 1015
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: TS_TypeDefs (p. 1037)

ASP_Constraints ::= SEQUENCE {

ttcn_ASP_Constraints	TTCN_ASP_Constraints		--See page 1038
asn1_ASP_Constraints	ASN1_ASP_Constraints		--See page 1010

}

Referenced from: ConstraintsPart (p. 1019)

ASP_ParDcl ::= SEQUENCE {

asp_ParId	ASP_ParId	FIELD	--See page 1045
asp_ParType	ASP_ParType	FIELD	--See page 1045
comment	Comment	FIELD	--See page 1097

}

Referenced from: ASP_ParDcls (p. 1015)

ASP_ParDcls ::= SEQUENCE OF {

asp_ParDcl	ASP_ParDcl		--See page 1015

}

Referenced from: TTCN_ASP_TypeDef (p. 1039)

ASP_ParValue ::= SEQUENCE {

asp_ParId	ASP_ParId	FIELD	--See page 1045
consValue	ConsValue	FIELD	--See page 1052
comment	Comment	FIELD	--See page 1097

}

Referenced from: ASP_ParValues (p. 1016)

ASP_ParValues ::= SEQUENCE OF {

asp_ParValue	ASP_ParValue	--See page 1016

}

Referenced from: TTCN_ASP_Constraint (p. 1038)

ASP_TypeDefs ::= SEQUENCE {

ttcn_ASP_TypeDefs	TTCN_ASP_TypeDefs	--See page 1039
asn1_ASP_TypeDefs	ASN1_ASP_TypeDefs	--See page 1011
asn1_ASP_TypeDefsByRef	ASN1_ASP_TypeDefsByRef	--See page 1011

}

Referenced from: ComplexDefinitions (p. 1019)

ASuite ::= SEQUENCE {

suiteId	SuiteId	FIELD	--See page 1083
suiteOverviewPart	SuiteOverviewPart		--See page 1030
declarationsPart	DeclarationsPart		--See page 1020
constraintsPart	ConstraintsPart		--See page 1019
dynamicPart	DynamicPart		--See page 1022

}

BehaviourDescription ::= SEQUENCE {

rootTree	RootTree	--See page 1027
localTreeList	LocalTreeList	--See page 1024

}

Referenced from: TestCase (p. 1032) TestStep (p. 1033) DefaultCase (p. 1020)

BehaviourLine ::= SEQUENCE {

lineNumber	LineNumber	FIELD	--See page 1063
labelId	LabelId	FIELD	--See page 1063
line	Line	FIELD	--See page 1063
cref	Cref	FIELD	--See page 1054
verdictId	VerdictId	FIELD	--See page 1094
comment	Comment	FIELD	--See page 1097
children	Children		--See page 1017

}

Referenced from: RootTree (p. 1027) BehaviourLineList (p. 1017) Children (p. 1017)

BehaviourLineList ::= SEQUENCE OF {

behaviourLine	BehaviourLine	--See page 1017

}

Referenced from: LocalTree (p. 1024)

Children ::= SEQUENCE OF {

behaviourLine	BehaviourLine	--See page 1017

}

Referenced from: BehaviourLine (p. 1017)

CM_Constraints ::= SEQUENCE {

ttcn_CM_Constraints	TTCN_CM_Constraints	--See page 1039
asn1_CM_Constraints	ASN1_CM_Constraints	--See page 1012

}

Referenced from: ConstraintsPart (p. 1019)

CM_ParDcl ::= SEQUENCE {

cm_ParId	CM_ParId	FIELD	--See page 1049
cm_ParType	CM_ParType	FIELD	--See page 1049
comment	Comment	FIELD	--See page 1097

}

Referenced from: CM_ParDcls (p. 1017)

CM_ParDcls ::= SEQUENCE OF {

cm_ParDcl	CM_ParDcl	--See page 1017

}

Referenced from: TTCN_CM_TypeDef (p. 1039)

CM_ParValue ::= SEQUENCE {

cm_ParId	CM_ParId	FIELD	--See page 1049
consValue	ConsValue	FIELD	--See page 1052
comment	Comment	FIELD	--See page 1097

}

Referenced from: CM_ParValues (p. 1018)

CM_ParValues ::= SEQUENCE OF {

cm_ParValue	CM_ParValue	--See page 1018

}

Referenced from: TTCN_CM_Constraint (p. 1039)

CM_TypeDefs ::= SEQUENCE {

ttcn_CM_TypeDefs	TTCN_CM_TypeDefs	--See page 1040
asn1_CM_TypeDefs	ASN1_CM_TypeDefs	--See page 1012

}

Referenced from: ComplexDefinitions (p. 1019)

CompactTestCase ::= SEQUENCE {

testCaseId	TestCaseId	FIELD	--See page 1085
testPurpose	TestPurpose	FIELD	--See page 1107
testStepAttachment	TestStepAttachment	FIELD	--See page 1085
comment	Comment	FIELD	--See page 1097
selExprId	SelExprId	FIELD	--See page 1078
description	Description	FIELD	--See page 1097

}

Referenced from: CompactTestCaseList (p. 1018)

CompactTestCaseList ::= SEQUENCE OF {

compactTestCase	CompactTestCase	--See page 1018

}

Referenced from: CompactTestGroup (p. 1019)

CompactTestGroup ::= SEQUENCE {

testGroupId	TestGroupId	FIELD	--See page 1085
defaultsRef	DefaultsRef	FIELD	--See page 1055
selExprId	SelExprId	OPTIONAL	--See page 1078
objective	Objective	OPTIONAL	--See page 1104
compactTestCaseList	CompactTestCaseList		--See page 1018

}

Referenced from: TestGroupOrTestCase (p. 1033)

ComplexDefinitions ::= SEQUENCE {

asp_TypeDefs	ASP_TypeDefs	--See page 1016
pdu_TypeDefs	PDU_TypeDefs	--See page 1026
cm_TypeDefs	CM_TypeDefs	--See page 1018
aliasDefs	AliasDefs	--See page 1010

}

Referenced from: DeclarationsPart (p. 1020)

ConstraintsPart ::= SEQUENCE {

ts_TypeConstraints	TS_TypeConstraints	--See page 1037
asp_Constraints	ASP_Constraints	--See page 1015
pdu_Constraints	PDU_Constraints	--See page 1026
cm_Constraints	CM_Constraints	--See page 1017

}

Referenced from: ASuite (p. 1016)

CP_Dcl ::= SEQUENCE {

cp_Id	CP_Id	FIELD	--See page 1053
comment	Comment	FIELD	--See page 1097

}

Referenced from: CP_DclList (p. 1019)

CP_DclList ::= SEQUENCE OF {

cp_Dcl	CP_Dcl	--See page 1019

}

Referenced from: CP_Dcls (p. 1020)

CP_Dcls ::= SEQUENCE {

cp_DclList	CP_DclList		--See page 1019
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: Declarations (p. 1020)

Declarations ::= SEQUENCE {

ts_ConstDcls	TS_ConstDcls	--See page 1035
ts_ConstRefs	TS_ConstRefs	--See page 1036
ts_VarDcls	TS_VarDcls	--See page 1038
tc_VarDcls	TC_VarDcls	--See page 1031
pco_TypeDcls	PCO_TypeDcls	--See page 1025
pco_Dcls	PCO_Dcls	--See page 1025
cp_Dcls	CP_Dcls	--See page 1020
timerDcls	TimerDcls	--See page 1035
tcompDcls	TCompDcls	--See page 1032
tcompConfigDcls	TCompConfigDcls	--See page 1031

}

Referenced from: DeclarationsPart (p. 1020)

DeclarationsPart ::= SEQUENCE {

definitions	Definitions	--See page 1021
paramAndSelection	ParamAndSelection	--See page 1024
declarations	Declarations	--See page 1020
complexDefinitions	ComplexDefinitions	--See page 1019

}

Referenced from: ASuite (p. 1016)

DefaultCase ::= SEQUENCE {

defaultId	DefaultId	FIELD	--See page 1054
defaultRef	DefaultRef	FIELD	--See page 1055
objective	Objective	FIELD	--See page 1104
comment	Comment	FIELD	--See page 1097
description	Description	FIELD	--See page 1097
behaviourDescription	BehaviourDescription		--See page 1016
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: DefaultGroupOrDefault (p. 1021)

DefaultGroup ::= SEQUENCE {

defaultGroupId	DefaultGroupId	FIELD	--See page 1054
defaultGroupOrDefaultList	DefaultGroupOrDefaultList		--See page 1021

}

Referenced from: DefaultGroupOrDefault (p. 1021)

DefaultGroupOrDefault ::= CHOICE {

defaultGroup	DefaultGroup	--See page 1021
defaultCase	DefaultCase	--See page 1020

}

Referenced from: DefaultGroupOrDefaultList (p. 1021)

DefaultGroupOrDefaultList ::= SEQUENCE OF {

defaultGroupOrDefault	DefaultGroupOrDefault	--See page 1021

}

Referenced from: DefaultsLibrary (p. 1021) DefaultGroup (p. 1021)

DefaultIndex ::= SEQUENCE {

SO_defIndexList	SO_DefIndexList		--See page 1028
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: SuiteOverviewPart (p. 1030)

DefaultsLibrary ::= SEQUENCE {

defaultGroupOrDefaultList	DefaultGroupOrDefaultList	--See page 1021

}

Referenced from: DynamicPart (p. 1022)

Definitions ::= SEQUENCE {

ts_TypeDefs	TS_TypeDefs	--See page 1037
encodingDefs	EncodingDefs	--See page 1023
ts_OpDefs	TS_OpDefs	--See page 1036
ts_ProcDefs	TS_ProcDefs	--See page 1037

}

Referenced from: DeclarationsPart (p. 1020)

DynamicPart ::= SEQUENCE {

testCases	TestCases		--See page 1033
testStepLibrary	TestStepLibrary		--See page 1034
defaultsLibrary	DefaultsLibrary		--See page 1021

}

Referenced from: ASuite (p. 1016)

ElemDcl ::= SEQUENCE {

elemId	ElemId	FIELD	--See page 1057
elemType	ElemType	FIELD	--See page 1057
pdu_FieldEncoding	PDU_FieldEncoding	FIELD	--See page 1071
comment	Comment	FIELD	--See page 1097

}

Referenced from: ElemDcls (p. 1022)

ElemDcls ::= SEQUENCE OF {

elemDcl	ElemDcl		--See page 1022

}

Referenced from: StructTypeDef (p. 1030)

ElemValue ::= SEQUENCE {

elemId	ElemId	FIELD	--See page 1057
consValue	ConsValue	FIELD	--See page 1052
comment	Comment	FIELD	--See page 1097

}

Referenced from: ElemValues (p. 1022)

ElemValues ::= SEQUENCE OF {

elemValue	ElemValue		--See page 1022

}

Referenced from: StructTypeConstraint (p. 1029)

EncodingDefinition ::= SEQUENCE {

encodingRuleId	EncodingRuleId	FIELD	--See page 1057
encodingRef	EncodingRef	FIELD	--See page 1057
encodingDefault	EncodingDefault	OPTIONAL	--See page 1057
comment	Comment	FIELD	--See page 1097

}

Referenced from: EncodingDefList (p. 1023)

EncodingDefinitions ::= SEQUENCE {

encodingDefList	EncodingDefList		--See page 1023
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: EncodingDefs (p. 1023)

EncodingDefList ::= SEQUENCE OF {

encodingDefinition	EncodingDefinition	--See page 1022

}

Referenced from: EncodingDefinitions (p. 1023)

EncodingDefs ::= SEQUENCE {

encodingDefinitions	EncodingDefinitions	--See page 1023
encodingVariations	EncodingVariations	--See page 1023
invalidFieldEncodingDefs	InvalidFieldEncodingDefs	--See page 1024

}

Referenced from: Definitions (p. 1021)

EncodingVariation ::= SEQUENCE {

encodingVariationId	EncodingVariationId	FIELD	--See page 1057
variationRef	VariationRef	FIELD	--See page 1093
variationDefault	VariationDefault	OPTIONAL	--See page 1093
comment	Comment	FIELD	--See page 1097

}

Referenced from: EncodingVariationList (p. 1023)

EncodingVariationList ::= SEQUENCE OF {

encodingVariation	EncodingVariation	--See page 1023

}

Referenced from: EncodingVariationSet (p. 1024)

EncodingVariations ::= SEQUENCE OF {

encodingVariationSet	EncodingVariationSet	--See page 1024

}

Referenced from: EncodingDefs (p. 1023)

EncodingVariationSet ::= SEQUENCE {

encodingRuleId	EncodingRuleId	FIELD	--See page 1057
encoding_TypeList	Encoding_TypeList	FIELD	--See page 1098
comment	Comment	FIELD	--See page 1097
encodingVariationList	EncodingVariationList		--See page 1023
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: EncodingVariations (p. 1023)

InvalidFieldEncodingDef ::= SEQUENCE {

invalidFieldEncodingId	InvalidFieldEncodingId	FIELD	--See page 1062
encoding_TypeList	Encoding_TypeList	FIELD	--See page 1098
comment	Comment	FIELD	--See page 1097
invalidFieldEncodingDefinition	InvalidFieldEncodingDefinitionFIELD		--See page 1101
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: InvalidFieldEncodingDefs (p. 1024)

InvalidFieldEncodingDefs ::= SEQUENCE OF {

invalidFieldEncodingDef	InvalidFieldEncodingDef	--See page 1024

}

Referenced from: EncodingDefs (p. 1023)

LocalTree ::= SEQUENCE {

header	Header	FIELD	--See page 1061
behaviourLineList	BehaviourLineList		--See page 1017

}

Referenced from: LocalTreeList (p. 1024)

LocalTreeList ::= SEQUENCE OF {

localTree	LocalTree	--See page 1024

}

Referenced from: BehaviourDescription (p. 1016)

ParamAndSelection ::= SEQUENCE {

ts_ParDcls	TS_ParDcls	--See page 1037
selectExprDefs	SelectExprDefs	--See page 1027

}

Referenced from: DeclarationsPart (p. 1020)

PCO_Dcl ::= SEQUENCE {

pco_Id	PCO_Id	FIELD	--See page 1070
pco_TypeId	PCO_TypeId	FIELD	--See page 1071
p_Role	P_Role	FIELD	--See page 1069
comment	Comment	FIELD	--See page 1097

}

Referenced from: PCO_DclList (p. 1025)

PCO_DclList ::= SEQUENCE OF {

pco_Dcl	PCO_Dcl		--See page 1025

}

Referenced from: PCO_Dcls (p. 1025)

PCO_Dcls ::= SEQUENCE {

pco_DclList	PCO_DclList		--See page 1025
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: Declarations (p. 1020)

PCO_TypeDcl ::= SEQUENCE {

pco_TypeId	PCO_TypeId	FIELD	--See page 1071
p_Role	P_Role	FIELD	--See page 1069
comment	Comment	FIELD	--See page 1097

}

Referenced from: PCO_TypeDclList (p. 1025)

PCO_TypeDclList ::= SEQUENCE OF {

pco_TypeDcl	PCO_TypeDcl		--See page 1025

}

Referenced from: PCO_TypeDcls (p. 1025)

PCO_TypeDcls ::= SEQUENCE {

pco_TypeDclList	PCO_TypeDclList		--See page 1025
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: Declarations (p. 1020)

PDU_Constraints ::= SEQUENCE {

ttcn_PDU_Constraints	TTCN_PDU_Constraints	--See page 1040
asn1_PDU_Constraints	ASN1_PDU_Constraints	--See page 1013

}

Referenced from: ConstraintsPart (p. 1019)

PDU_FieldDcl ::= SEQUENCE {

pdu_FieldId	PDU_FieldId	FIELD	--See page 1072
pdu_FieldType	PDU_FieldType	FIELD	--See page 1072
pdu_FieldEncoding	PDU_FieldEncoding	FIELD	--See page 1071
comment	Comment	FIELD	--See page 1097

}

Referenced from: PDU_FieldDcls (p. 1026)

PDU_FieldDcls ::= SEQUENCE OF {

pdu_FieldDcl	PDU_FieldDcl	--See page 1026

}

Referenced from: TTCN_PDU_TypeDef (p. 1040)

PDU_FieldValue ::= SEQUENCE {

pdu_FieldId	PDU_FieldId	FIELD	--See page 1072
consValue	ConsValue	FIELD	--See page 1052
pdu_FieldEncoding	PDU_FieldEncoding	FIELD	--See page 1071
comment	Comment	FIELD	--See page 1097

}

Referenced from: PDU_FieldValues (p. 1026)

PDU_FieldValues ::= SEQUENCE OF {

pdu_FieldValue	PDU_FieldValue	--See page 1026

}

Referenced from: TTCN_PDU_Constraint (p. 1040)

PDU_TypeDefs ::= SEQUENCE {

ttcn_PDU_TypeDefs	TTCN_PDU_TypeDefs	--See page 1040
asn1_PDU_TypeDefs	ASN1_PDU_TypeDefs	--See page 1013
asn1_PDU_TypeDefsByRef	ASN1_PDU_TypeDefsByRef	--See page 1014

}

Referenced from: ComplexDefinitions (p. 1019)

RootTree ::= SEQUENCE OF {

behaviourLine	BehaviourLine		--See page 1017

}

Referenced from: BehaviourDescription (p. 1016)

SelectExprDef ::= SEQUENCE {

selectExprId	SelectExprId	FIELD	--See page 1077
selectExpr	SelectExpr	FIELD	--See page 1077
comment	Comment	FIELD	--See page 1097

}

Referenced from: SelectExprDefList (p. 1027)

SelectExprDefList ::= SEQUENCE OF {

selectExprDef	SelectExprDef		--See page 1027

}

Referenced from: SelectExprDefs (p. 1027)

SelectExprDefs ::= SEQUENCE {

selectExprDefList	SelectExprDefList		--See page 1027
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: ParamAndSelection (p. 1024)

SimpleTypeDef ::= SEQUENCE {

simpleTypeId	SimpleTypeId	FIELD	--See page 1080
simpleTypeDefinition	SimpleTypeDefinition	FIELD	--See page 1080
pdu_FieldEncoding	PDU_FieldEncoding	FIELD	--See page 1071
comment	Comment	FIELD	--See page 1097

}

Referenced from: SimpleTypeDefList (p. 1027)

SimpleTypeDefList ::= SEQUENCE OF {

simpleTypeDef	SimpleTypeDef		--See page 1027

}

Referenced from: SimpleTypeDefs (p. 1028)

SimpleTypeDefs ::= SEQUENCE {

simpleTypeDefList	SimpleTypeDefList		--See page 1027
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: TS_TypeDefs (p. 1037)

SO_CaseIndex ::= SEQUENCE {

SO_testGroupRef	SO_TestGroupRef	FIELD	--See page 1107
SO_testCaseId	SO_TestCaseId	FIELD	--See page 1107
SO_selExprId	SO_SelExprId	FIELD	--See page 1106
SO_description	SO_Description	FIELD	--See page 1106

}

Referenced from: SO_CaseIndexList (p. 1028)

SO_CaseIndexList ::= SEQUENCE OF {

SO_caseIndex	SO_CaseIndex		--See page 1028

}

Referenced from: TestCaseIndex (p. 1032)

SO_DefIndex ::= SEQUENCE {

SO_defaultRef	SO_DefaultRef	FIELD	--See page 1106
SO_defaultId	SO_DefaultId	FIELD	--See page 1106
SO_description	SO_Description	FIELD	--See page 1106

}

Referenced from: SO_DefIndexList (p. 1028)

SO_DefIndexList ::= SEQUENCE OF {

SO_defIndex	SO_DefIndex		--See page 1028

}

Referenced from: DefaultIndex (p. 1021)

SO_StepIndex ::= SEQUENCE {

SO_testStepRef	SO_TestStepRef	FIELD	--See page 1107
SO_testStepId	SO_TestStepId	FIELD	--See page 1107
SO_description	SO_Description	FIELD	--See page 1106

}

Referenced from: SO_StepIndexList (p. 1029)

SO_StepIndexList ::= SEQUENCE OF {

SO_stepIndex	SO_StepIndex		--See page 1028

}

Referenced from: TestStepIndex (p. 1034)

SO_StructureAndObjective ::= SEQUENCE {

SO_testGroupRef	SO_TestGroupRef	FIELD	--See page 1107
SO_selExprId	SO_SelExprId	FIELD	--See page 1106
SO_objective	SO_Objective	FIELD	--See page 1106

}

Referenced from: SO_StructureAndObjectives (p. 1029)

SO_StructureAndObjectives ::= SEQUENCE OF {

SO_structureAndObjective	SO_StructureAndObjective		--See page 1029

}

Referenced from: SuiteStructure (p. 1030)

StructTypeConstraint ::= SEQUENCE {

consId	ConsId	FIELD	--See page 1050
structId	StructId	FIELD	--See page 1082
derivPath	DerivPath	FIELD	--See page 1056
encVariationId	EncVariationId	FIELD	--See page 1058
comment	Comment	FIELD	--See page 1097
elemValues	ElemValues		--See page 1022
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: StructTypeConstraints (p. 1029)

StructTypeConstraints ::= SEQUENCE OF {

structTypeConstraint	StructTypeConstraint		--See page 1029

}

Referenced from: TS_TypeConstraints (p. 1037)

StructTypeDef ::= SEQUENCE {

structId	StructId	FIELD	--See page 1082
encVariationId	EncVariationId	FIELD	--See page 1058
comment	Comment	FIELD	--See page 1097
elemDcls	ElemDcls		--See page 1022
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: StructTypeDefs (p. 1030)

StructTypeDefs ::= SEQUENCE OF {

structTypeDef	StructTypeDef		--See page 1030

}

Referenced from: TS_TypeDefs (p. 1037)

SuiteOverviewPart ::= SEQUENCE {

suiteStructure	SuiteStructure		--See page 1030
testCaseIndex	TestCaseIndex		--See page 1032
testStepIndex	TestStepIndex		--See page 1034
defaultIndex	DefaultIndex		--See page 1021

}

Referenced from: ASuite (p. 1016)

SuiteStructure ::= SEQUENCE {

SO_suiteId	SO_SuiteId	FIELD	--See page 1081
SO_standardsRef	SO_StandardsRef	FIELD	--See page 1107
SO_picsRef	SO_PICSRef	FIELD	--See page 1106
SO_pixitRef	SO_PIXITRef	FIELD	--See page 1106
SO_testMethods	SO_TestMethods	FIELD	--See page 1107
comment	Comment	FIELD	--See page 1097
SO_structureAndObjectives	SO_StructureAndObjectives		--See page 1029
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: SuiteOverviewPart (p. 1030)

TC_VarDcl ::= SEQUENCE {

tc_VarId	TC_VarId	FIELD	--See page 1083
tc_VarType	TC_VarType	FIELD	--See page 1084
tc_VarValue	TC_VarValue	FIELD	--See page 1084
comment	Comment	FIELD	--See page 1097

}

Referenced from: TC_VarDclList (p. 1031)

TC_VarDclList ::= SEQUENCE OF {

tc_VarDcl	TC_VarDcl		--See page 1030

}

Referenced from: TC_VarDcls (p. 1031)

TC_VarDcls ::= SEQUENCE {

tc_VarDclList	TC_VarDclList		--See page 1031
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: Declarations (p. 1020)

TCompConfigDcl ::= SEQUENCE {

tcompConfigId	TCompConfigId	FIELD	--See page 1084
comment	Comment	FIELD	--See page 1097
tcompConfigInfoList	TCompConfigInfoList		--See page 1031
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: TCompConfigDcls (p. 1031)

TCompConfigDcls ::= SEQUENCE OF {

tcompConfigDcl	TCompConfigDcl		--See page 1031

}

Referenced from: Declarations (p. 1020)

TCompConfigInfo ::= SEQUENCE {

tcompUsed	TCompUsed	FIELD	--See page 1084
pcos_Used	PCOs_Used	FIELD	--See page 1071
cps_Used	CPs_Used	FIELD	--See page 1053
comment	Comment	FIELD	--See page 1097

}

Referenced from: TCompConfigInfoList (p. 1031)

TCompConfigInfoList ::= SEQUENCE OF {

tcompConfigInfo	TCompConfigInfo		--See page 1031

}

Referenced from: TCompConfigDcl (p. 1031)

TCompDcl ::= SEQUENCE {

tcompId	TCompId	FIELD	--See page 1084
tc_Role	TC_Role	FIELD	--See page 1083
numOf_PCOs	NumOf_PCOs	FIELD	--See page 1068
numOf_CPs	NumOf_CPs	FIELD	--See page 1068
comment	Comment	FIELD	--See page 1097

}

Referenced from: TCompDclList (p. 1032)

TCompDclList ::= SEQUENCE OF {

tcompDcl	TCompDcl		--See page 1032

}

Referenced from: TCompDcls (p. 1032)

TCompDcls ::= SEQUENCE {

tcompDclList	TCompDclList		--See page 1032
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: Declarations (p. 1020)

TestCase ::= SEQUENCE {

testCaseId	TestCaseId	FIELD	--See page 1085
testGroupRef	TestGroupRef	FIELD	--See page 1085
testPurpose	TestPurpose	FIELD	--See page 1107
configuration	Configuration	OPTIONAL	--See page 1050
defaultsRef	DefaultsRef	OPTIONAL	--See page 1055
comment	Comment	FIELD	--See page 1097
selExprId	SelExprId	FIELD	--See page 1078
description	Description	FIELD	--See page 1097
behaviourDescription	BehaviourDescription		--See page 1016
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: TestGroupOrTestCase (p. 1033)

TestCaseIndex ::= SEQUENCE {

SO_caseIndexList	SO_CaseIndexList		--See page 1028
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: SuiteOverviewPart (p. 1030)

TestCases ::= SEQUENCE {

testGroupOrTestCaseList	TestGroupOrTestCaseList		--See page 1033

}

Referenced from: DynamicPart (p. 1022)

TestGroup ::= SEQUENCE {

testGroupId	TestGroupId	FIELD	--See page 1085
selExprId	SelExprId	OPTIONAL	--See page 1078
objective	Objective	OPTIONAL	--See page 1104
testGroupOrTestCaseList	TestGroupOrTestCaseList		--See page 1033

}

Referenced from: TestGroupOrTestCase (p. 1033)

TestGroupOrTestCase ::= CHOICE {

testGroup	TestGroup	--See page 1033
testCase	TestCase	--See page 1032
compactTestGroup	CompactTestGroup	--See page 1019

}

Referenced from: TestGroupOrTestCaseList (p. 1033)

TestGroupOrTestCaseList ::= SEQUENCE OF {

testGroupOrTestCase	TestGroupOrTestCase	--See page 1033

}

Referenced from: TestCases (p. 1033) TestGroup (p. 1033)

TestStep ::= SEQUENCE {

testStepId	TestStepId	FIELD	--See page 1085
testStepRef	TestStepRef	FIELD	--See page 1086
objective	Objective	FIELD	--See page 1104
defaultsRef	DefaultsRef	FIELD	--See page 1055
comment	Comment	FIELD	--See page 1097
description	Description	FIELD	--See page 1097
behaviourDescription	BehaviourDescription		--See page 1016
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: TestStepGroupOrTestStep (p. 1034)

TestStepGroup ::= SEQUENCE {

testStepGroupId	TestStepGroupId	FIELD	--See page 1085
testStepGroupOrTestStepList	TestStepGroupOrTestStepList		--See page 1034

}

Referenced from: TestStepGroupOrTestStep (p. 1034)

TestStepGroupOrTestStep ::= CHOICE {

testStepGroup	TestStepGroup	--See page 1034
testStep	TestStep	--See page 1033

}

Referenced from: TestStepGroupOrTestStepList (p. 1034)

TestStepGroupOrTestStepList ::= SEQUENCE OF {

testStepGroupOrTestStep	TestStepGroupOrTestStep	--See page 1034

}

Referenced from: TestStepLibrary (p. 1034) TestStepGroup (p. 1034)

TestStepIndex ::= SEQUENCE {

SO_stepIndexList	SO_StepIndexList		--See page 1029
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: SuiteOverviewPart (p. 1030)

TestStepLibrary ::= SEQUENCE {

testStepGroupOrTestStepList	TestStepGroupOrTestStepList		--See page 1034

}

Referenced from: DynamicPart (p. 1022)

TimerDcl ::= SEQUENCE {

timerId	TimerId	FIELD	--See page 1086
duration	Duration	FIELD	--See page 1056
unit	Unit	FIELD	--See page 1091
comment	Comment	FIELD	--See page 1097

}

Referenced from: TimerDclList (p. 1035)

TimerDclList ::= SEQUENCE OF {

timerDcl	TimerDcl		--See page 1034

}

Referenced from: TimerDcls (p. 1035)

TimerDcls ::= SEQUENCE {

timerDclList	TimerDclList		--See page 1035
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: Declarations (p. 1020)

TS_ConstDcl ::= SEQUENCE {

ts_ConstId	TS_ConstId	FIELD	--See page 1087
ts_ConstType	TS_ConstType	FIELD	--See page 1087
ts_ConstValue	TS_ConstValue	FIELD	--See page 1087
comment	Comment	FIELD	--See page 1097

}

Referenced from: TS_ConstDclList (p. 1035)

TS_ConstDclList ::= SEQUENCE OF {

ts_ConstDcl	TS_ConstDcl		--See page 1035

}

Referenced from: TS_ConstDcls (p. 1035)

TS_ConstDcls ::= SEQUENCE {

ts_ConstDclList	TS_ConstDclList		--See page 1035
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: Declarations (p. 1020)

TS_ConstRef ::= SEQUENCE {

ts_ConstId	TS_ConstId	FIELD	--See page 1087
ts_ConstType	TS_ConstType	FIELD	--See page 1087
asn1_ValueReference	ASN1_ValueReference	FIELD	--See page 1096
asn1_ModuleId	ASN1_ModuleId	FIELD	--See page 1096
comment	Comment	FIELD	--See page 1097
ts_ConstValue	TS_ConstValue	OPTIONAL	--See page 1087

}

Referenced from: TS_ConstRefList (p. 1036)

TS_ConstRefList ::= SEQUENCE OF {

ts_ConstRef	TS_ConstRef		--See page 1035

}

Referenced from: TS_ConstRefs (p. 1036)

TS_ConstRefs ::= SEQUENCE {

ts_ConstRefList	TS_ConstRefList		--See page 1036
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: Declarations (p. 1020)

TS_OpDef ::= SEQUENCE {

ts_OpId	TS_OpId	FIELD	--See page 1088
ts_OpResult	TS_OpResult	FIELD	--See page 1088
comment	Comment	FIELD	--See page 1097
ts_OpDescription	TS_OpDescription	FIELD	--See page 1108
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: TS_OpDefs (p. 1036)

TS_OpDefs ::= SEQUENCE OF {

ts_OpDef	TS_OpDef		--See page 1036

}

Referenced from: Definitions (p. 1021)

TS_ParDcl ::= SEQUENCE {

ts_ParId	TS_ParId	FIELD	--See page 1088
ts_ParType	TS_ParType	FIELD	--See page 1088
pics_PIXITref	PICS_PIXITref	FIELD	--See page 1105
comment	Comment	FIELD	--See page 1097

}

Referenced from: TS_ParDclList (p. 1036)

TS_ParDclList ::= SEQUENCE OF {

ts_ParDcl	TS_ParDcl		--See page 1036

}

Referenced from: TS_ParDcls (p. 1037)

TS_ParDcls ::= SEQUENCE {

ts_ParDclList	TS_ParDclList		--See page 1036
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: ParamAndSelection (p. 1024)

TS_ProcDef ::= SEQUENCE {

ts_ProcId	TS_ProcId	FIELD	--See page 1088
ts_ProcResult	TS_ProcResult	FIELD	--See page 1089
comment	Comment	FIELD	--See page 1097
ts_ProcDescription	TS_ProcDescription	FIELD	--See page 1108
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: TS_ProcDefs (p. 1037)

TS_ProcDefs ::= SEQUENCE OF {

ts_ProcDef	TS_ProcDef	--See page 1037

}

Referenced from: Definitions (p. 1021)

TS_TypeConstraints ::= SEQUENCE {

structTypeConstraints	StructTypeConstraints	--See page 1029
asn1_TypeConstraints	ASN1_TypeConstraints	--See page 1014

}

Referenced from: ConstraintsPart (p. 1019)

TS_TypeDefs ::= SEQUENCE {

simpleTypeDefs	SimpleTypeDefs	--See page 1028
structTypeDefs	StructTypeDefs	--See page 1030
asn1_TypeDefs	ASN1_TypeDefs	--See page 1014
asn1_TypeRefs	ASN1_TypeRefs	--See page 1015

}

Referenced from: Definitions (p. 1021)

TS_VarDcl ::= SEQUENCE {

ts_VarId	TS_VarId	FIELD	--See page 1089
ts_VarType	TS_VarType	FIELD	--See page 1089
ts_VarValue	TS_VarValue	OPTIONAL	--See page 1089
comment	Comment	FIELD	--See page 1097

}

Referenced from: TS_VarDclList (p. 1038)

TS_VarDclList ::= SEQUENCE OF {

ts_VarDcl	TS_VarDcl		--See page 1038

}

Referenced from: TS_VarDcls (p. 1038)

TS_VarDcls ::= SEQUENCE {

ts_VarDclList	TS_VarDclList		--See page 1038
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: Declarations (p. 1020)

TTCN_ASP_Constraint ::= SEQUENCE {

consId	ConsId	FIELD	--See page 1050
asp_Id	ASP_Id	FIELD	--See page 1044
derivPath	DerivPath	FIELD	--See page 1056
comment	Comment	FIELD	--See page 1097
asp_ParValues	ASP_ParValues		--See page 1016
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: TTCN_ASP_Constraints (p. 1038)

TTCN_ASP_Constraints ::= SEQUENCE OF {

ttcn_ASP_Constraint	TTCN_ASP_Constraint		--See page 1038

}

Referenced from: ASP_Constraints (p. 1015)

TTCN_ASP_TypeDef ::= SEQUENCE {

asp_Id	ASP_Id	FIELD	--See page 1044
pco_Type	PCO_Type	FIELD	--See page 1071
comment	Comment	FIELD	--See page 1097
asp_ParDcls	ASP_ParDcls		--See page 1015
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: TTCN_ASP_TypeDefs (p. 1039)

TTCN_ASP_TypeDefs ::= SEQUENCE OF {

ttcn_ASP_TypeDef	TTCN_ASP_TypeDef		--See page 1039

}

Referenced from: ASP_TypeDefs (p. 1016)

TTCN_CM_Constraint ::= SEQUENCE {

consId	ConsId	FIELD	--See page 1050
cm_Id	CM_Id	FIELD	--See page 1049
derivPath	DerivPath	FIELD	--See page 1056
comment	Comment	FIELD	--See page 1097
cm_ParValues	CM_ParValues		--See page 1018
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: TTCN_CM_Constraints (p. 1039)

TTCN_CM_Constraints ::= SEQUENCE OF {

ttcn_CM_Constraint	TTCN_CM_Constraint		--See page 1039

}

Referenced from: CM_Constraints (p. 1017)

TTCN_CM_TypeDef ::= SEQUENCE {

cm_Id	CM_Id	FIELD	--See page 1049
comment	Comment	FIELD	--See page 1097
cm_ParDcls	CM_ParDcls		--See page 1017
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: TTCN_CM_TypeDefs (p. 1040)

TTCN_CM_TypeDefs ::= SEQUENCE OF {

ttcn_CM_TypeDef	TTCN_CM_TypeDef		--See page 1039

}

Referenced from: CM_TypeDefs (p. 1018)

TTCN_PDU_Constraint ::= SEQUENCE {

consId	ConsId	FIELD	--See page 1050
pdu_Id	PDU_Id	FIELD	--See page 1072
derivPath	DerivPath	FIELD	--See page 1056
encRuleId	EncRuleId	FIELD	--See page 1058
encVariationId	EncVariationId	FIELD	--See page 1058
comment	Comment	FIELD	--See page 1097
pdu_FieldValues	PDU_FieldValues		--See page 1026
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: TTCN_PDU_Constraints (p. 1040)

TTCN_PDU_Constraints ::= SEQUENCE OF {

ttcn_PDU_Constraint	TTCN_PDU_Constraint		--See page 1040

}

Referenced from: PDU_Constraints (p. 1026)

TTCN_PDU_TypeDef ::= SEQUENCE {

pdu_Id	PDU_Id	FIELD	--See page 1072
pco_Type	PCO_Type	FIELD	--See page 1071
pdu_EncodingId	PDU_EncodingId	FIELD	--See page 1071
encVariationId	EncVariationId	FIELD	--See page 1058
comment	Comment	FIELD	--See page 1097
pdu_FieldDcls	PDU_FieldDcls		--See page 1026
detailedComment	DetailedComment	FIELD	--See page 1098

}

Referenced from: TTCN_PDU_TypeDefs (p. 1040)

TTCN_PDU_TypeDefs ::= SEQUENCE OF {

ttcn_PDU_TypeDef	TTCN_PDU_TypeDef		--See page 1040

}

Referenced from: PDU_TypeDefs (p. 1026)

Parse Tree Nodes

This section contains all nodes within the Parse Trees. The Parse Trees are the output of the Analyzer, and they are a representation of whatever code the Test Suite writer put into the fields. Being user written, this part is much less well behaved than, say, the Static part, nodes may or may not be present, fields may or may not be well analyzed and some nodes have different children in different places in the Parse Tree (choices). All this is well specified however, using the mechanisms of is_present, is_analyzed (for tables) and the choice member function, but it is up to the user of TTCN Access to be aware of these constructs.

Activate ::= SEQUENCE {

defaultRefList	DefaultRefList	--See page 1055

}

Referenced from: Construct (p. 1052)

ActualCrefParList ::= SEQUENCE OF {

constraintValueAndAttributes	ConstraintValueAndAttributes	--See page 1052

}

Referenced from: ConsRef (p. 1051)

ActualParList ::= SEQUENCE OF {

constraintValueAndAttributes	ConstraintValueAndAttributes	--See page 1052

}

Referenced from: EncVariationCall (p. 1058) InvalidFieldEncodingCall (p. 1062) DefaultReference (p. 1055) Attach (p. 1045) CreateAttach (p. 1053) OpCall (p. 1069)

AddExpression ::= SEQUENCE {

simpleExpression	SimpleExpression	--See page 1079
addOp	AddOp	--See page 1041
term	Term	--See page 1085

}

Referenced from: SimpleExpression (p. 1079)

AddOp ::= CHOICE {

plus	Plus	--See page 1105
minus	Minus	--See page 1102
or	Or	--See page 1104

}

Referenced from: AddExpression (p. 1041)

AliasId ::= SEQUENCE {

aliasIdentifier	Identifier	--See page 1100

}

Referenced from: AliasDef (p. 1010)

AlternativeTypeList ::= SEQUENCE OF {

namedType	NamedType	--See page 1067

}

Referenced from: ChoiceType (p. 1048)

AnyDefinedBy ::= SEQUENCE {

identifier	Identifier	--See page 1100

}

Referenced from: AnyType (p. 1042)

AnyType ::= CHOICE {

any	ANY	--See page 1095
anyDefinedBy	AnyDefinedBy	--See page 1042

}

Referenced from: BuiltinType (p. 1047)

AnyValue ::= SEQUENCE {

referenceType	ReferenceType	--See page 1075
constraintValueAndAttributes	ConstraintValueAndAttributes	--See page 1052

}

Referenced from: ASN1_Value (p. 1044)

ArrayRef ::= SEQUENCE {

componentNumber	ComponentNumber	--See page 1050

}

Referenced from: ComponentReference (p. 1050)

ArrayRefOrComp ::= CHOICE {

componentIdentifier	ComponentIdentifier	--See page 1049
bitRef	BitRef	--See page 1046
componentPosition	ComponentPosition	--See page 1050

}

Referenced from: ReferenceList (p. 1075)

ASN1_ConsValue ::= SEQUENCE {
constraintValueAndAttributesOrReplaceConstraintValueAndAttributesOrReplace--See page 1052
}
Referenced from: ASN1_TypeConstraint (p. 1014) ASN1_ASP_Constraint (p. 1010) ASN1_PDU_Constraint (p. 1012) ASN1_CM_Constraint (p. 1012)

ASN1_LocalTypes ::= SEQUENCE {

typeAssignmentList	TypeAssignmentList	--See page 1090

}
Referenced from: ASN1_TypeAndLocalTypes (p. 1043)

ASN1_Type ::= CHOICE {

builtinType	BuiltinType	--See page 1047
definedType	DefinedType	--See page 1055
subType	SubType	--See page 1082

}
Referenced from: ASN1_TypeAndLocalTypes (p. 1043) ComponentsOf (p. 1050) NamedType (p. 1067) SequenceOfType (p. 1078) SetOfType (p. 1079) SelectionType (p. 1077) TaggedType (p. 1083) ContainedSubType (p. 1052) ParentType (p. 1070) SetSubType (p. 1079) SequenceSubType (p. 1078) TypeAssignment (p. 1090)

ASN1_TypeAndLocalTypes ::= SEQUENCE {

asn1_Type	ASN1_Type	--See page 1043
asn1_LocalTypes	ASN1_LocalTypes	--See page 1043

}
Referenced from: ASN1_TypeDefinition (p. 1043)

ASN1_TypeDefinition ::= SEQUENCE {

asn1_TypeAndLocalTypes	ASN1_TypeAndLocalTypes	--See page 1043

}
Referenced from: ASN1_TypeDef (p. 1014) ASN1_TypeRef (p. 1015) ASN1_ASP_TypeDef (p. 1011) ASN1_ASP_TypeDefByRef (p. 1011) ASN1_PDU_TypeDef (p. 1013) ASN1_PDU_TypeDefByRef (p. 1013) ASN1_CM_TypeDef (p. 1012)

ASN1_TypeId ::= SEQUENCE {

asn1_TypeIdentifier	Identifier		--See page 1100
fullIdentifier	FullIdentifier	OPTIONAL	--See page 1099

}

Referenced from: ASN1_TypeDef (p. 1014) ASN1_TypeRef (p. 1015) ASN1_TypeConstraint (p. 1014)

ASN1_Value ::= CHOICE {

bitStringValue	BitStringValue	--See page 1046
nullValue	NullValue	--See page 1103
sequenceValue	SequenceValue	--See page 1078
sequenceOfValue	SequenceOfValue	--See page 1078
setValue	SetValue	--See page 1079
setOfValue	SetOfValue	--See page 1079
choiceValue	ChoiceValue	--See page 1048
selectionValue	SelectionValue	--See page 1077
anyValue	AnyValue	--See page 1042
objectIdentifierValue	ObjectIdentifierValue	--See page 1068
realValue	RealValue	--See page 1075

}

Referenced from: Value (p. 1092)

ASN1_ValueList ::= SEQUENCE OF {

constraintValueAndAttributes	ConstraintValueAndAttributes	--See page 1052

}

Referenced from: SequenceOfValue (p. 1078) SetOfValue (p. 1079)

ASP_Id ::= SEQUENCE {

asp_IdAndFullId	ASP_IdAndFullId	--See page 1044

}

Referenced from: TTCN_ASP_TypeDef (p. 1039) ASN1_ASP_TypeDef (p. 1011) ASN1_ASP_TypeDefByRef (p. 1011) TTCN_ASP_Constraint (p. 1038) ASN1_ASP_Constraint (p. 1010)

ASP_IdAndFullId ::= SEQUENCE {

asp_Identifier	Identifier		--See page 1100
fullIdentifier	FullIdentifier	OPTIONAL	--See page 1099

}

Referenced from: ASP_Id (p. 1044)

ASP_ParId ::= SEQUENCE {

asp_ParIdOrMacro	ASP_ParIdOrMacro		--See page 1045

}

Referenced from: ASP_ParDcl (p. 1015) ASP_ParValue (p. 1016)

ASP_ParIdAndFullId ::= SEQUENCE {

asp_ParIdentifier	Identifier		--See page 1100
fullIdentifier	FullIdentifier	OPTIONAL	--See page 1099

}

Referenced from: ASP_ParIdOrMacro (p. 1045)

ASP_ParIdOrMacro ::= CHOICE {

asp_ParIdAndFullId	ASP_ParIdAndFullId	--See page 1045
macroSymbol	MacroSymbol	--See page 1102

}

Referenced from: ASP_ParId (p. 1045)

ASP_ParType ::= SEQUENCE {

typeAndAttributes	TypeAndAttributes	--See page 1089

}

Referenced from: ASP_ParDcl (p. 1015)

Assignment ::= SEQUENCE {

dataObjectReference	DataObjectReference	--See page 1054
expression	Expression	--See page 1060

}

Referenced from: AssignmentList (p. 1045)

AssignmentList ::= SEQUENCE OF {

assignment	Assignment	--See page 1045

}

Referenced from: EventStatement (p. 1059)

Attach ::= SEQUENCE {

treeReference	TreeReference	--See page 1087
actualParList	ActualParList	--See page 1041

}

Referenced from: TestStepAttachment (p. 1085) Construct (p. 1052) Repeat (p. 1076)

Base ::= CHOICE {
    two   Two   --See page 1108
    ten   Ten   --See page 1107
}
Referenced from: NumericRealValue (p. 1068)

BitIdentifier ::= SEQUENCE {
    identifier   Identifier   --See page 1100
}
Referenced from: BitRef (p. 1046)

BitNumber ::= SEQUENCE {
    expression   Expression   --See page 1060
}
Referenced from: BitRef (p. 1046)

BitRef ::= CHOICE {
    bitIdentifier   BitIdentifier   --See page 1046
    bitNumber   BitNumber   --See page 1046
}
Referenced from: ArrayRefOrComp (p. 1042) ComponentReference (p. 1050)

BitStringType ::= SEQUENCE {
    namedBitList   NamedBitList   --See page 1066
}
Referenced from: BuiltinType (p. 1047)

BitStringValue ::= SEQUENCE {
    identifierList   IdentifierList   --See page 1061
}
Referenced from: ASN1_Value (p. 1044)

BooleanValue ::= CHOICE {
    Atrue   ATrue   --See page 1096
    Afalse   AFalse   --See page 1095
}
Referenced from: LiteralValue (p. 1064)

Bound ::= CHOICE {

number	Number	--See page 1103
identifier	Identifier	--See page 1100

}

Referenced from: SingleLength (p. 1080) LowerBound (p. 1064) UpperBound (p. 1091)

BuiltinType ::= CHOICE {

booleanType	BooleanType	--See page 1096
integerType	IntegerType	--See page 1062
bitStringType	BitStringType	--See page 1046
octetStringType	OctetStringType	--See page 1104
nullType	NullType	--See page 1103
sequenceType	SequenceType	--See page 1078
sequenceOfType	SequenceOfType	--See page 1078
setType	SetType	--See page 1079
setOfType	SetOfType	--See page 1079
choiceType	ChoiceType	--See page 1048
selectionType	SelectionType	--See page 1077
taggedType	TaggedType	--See page 1083
anyType	AnyType	--See page 1042
objectIdentifierType	ObjectIdentifierType	--See page 1104
characterString	CharacterString	--See page 1048
usefulType	UsefulType	--See page 1092
enumeratedType	EnumeratedType	--See page 1058
realType	RealType	--See page 1106

}

Referenced from: ASN1_Type (p. 1043)

CancelTimer ::= SEQUENCE {

timerIdentifier	Identifier	OPTIONAL	--See page 1100

}

Referenced from: TimerOp (p. 1086)

CharacterString ::= CHOICE {

numericString	NumericString	--See page 1103
printableString	PrintableString	--See page 1105
teletexString	TeletexString	--See page 1107
videotexString	VideotexString	--See page 1108
visibleString	VisibleString	--See page 1108
ia5String	IA5String	--See page 1099
graphicString	GraphicString	--See page 1099
generalString	GeneralString	--See page 1099
t61String	T61String	--See page 1107
iso646String	ISO646String	--See page 1101

}

Referenced from: PredefinedType (p. 1073) BuiltinType (p. 1047)

ChoiceType ::= SEQUENCE {

alternativeTypeList	AlternativeTypeList	--See page 1042

}

Referenced from: BuiltinType (p. 1047)

ChoiceValue ::= SEQUENCE {

namedValue	NamedValue	--See page 1067

}

Referenced from: ASN1_Value (p. 1044)

ClassNumber ::= CHOICE {

number	Number	--See page 1103
definedValue	DefinedValue	--See page 1055

}

Referenced from: Tag (p. 1083)

ClassType ::= CHOICE {

universal	UNIVERSAL	--See page 1108
application	APPLICATION	--See page 1096
private_	PRIVATE_	--See page 1105

}

Referenced from: Tag (p. 1083)

CM_Id ::= SEQUENCE {

cm_Identifier	Identifier		--See page 1100

}

Referenced from: TTCN_CM_TypeDef (p. 1039) ASN1_CM_TypeDef (p. 1012)
TTCN_CM_Constraint (p. 1039) ASN1_CM_Constraint (p. 1012)

CM_ParId ::= SEQUENCE {

cm_ParIdOrMacro	CM_ParIdOrMacro		--See page 1049

}

Referenced from: CM_ParDcl (p. 1017) CM_ParValue (p. 1018)

CM_ParIdAndFullId ::= SEQUENCE {

cm_ParIdentifier	Identifier		--See page 1100
fullIdentifier	FullIdentifier	OPTIONAL	--See page 1099

}

Referenced from: CM_ParIdOrMacro (p. 1049)

CM_ParIdOrMacro ::= CHOICE {

cm_ParIdAndFullId	CM_ParIdAndFullId	--See page 1049
macroSymbol	MacroSymbol	--See page 1102

}

Referenced from: CM_ParId (p. 1049)

CM_ParType ::= SEQUENCE {

typeAndAttributes	TypeAndAttributes	--See page 1089

}

Referenced from: CM_ParDcl (p. 1017)

Complement ::= SEQUENCE {

valueList	ValueList	--See page 1093

}

Referenced from: MatchingSymbol (p. 1065)

ComponentIdentifier ::= SEQUENCE {

identifier	Identifier	--See page 1100

}

Referenced from: ArrayRefOrComp (p. 1042) RecordRef (p. 1075)

ComponentNumber ::= SEQUENCE {
    expression Expression --See page 1060
}
Referenced from: ArrayRef (p. 1042)

ComponentPosition ::= SEQUENCE {
    number Number --See page 1103
}
Referenced from: ArrayRefOrComp (p. 1042)

ComponentReference ::= CHOICE {
    recordRef RecordRef --See page 1075
    arrayRef ArrayRef --See page 1042
    bitRef BitRef --See page 1046
}
Referenced from: ComponentReferenceList (p. 1050)

ComponentReferenceList ::= SEQUENCE OF {
    componentReference ComponentReference --See page 1050
}
Referenced from: ReferenceList (p. 1075) DataObjectReference (p. 1054)

ComponentsOf ::= SEQUENCE {
    asn1_Type ASN1_Type --See page 1043
}
Referenced from: ElementType (p. 1056)

Configuration ::= SEQUENCE {
    tcompConfigIdentifier Identifier --See page 1100
}
Referenced from: TestCase (p. 1032)

ConsId ::= SEQUENCE {
    consIdAndParList ConsIdAndParList --See page 1051
}
Referenced from: StructTypeConstraint (p. 1029) ASN1_TypeConstraint (p. 1014)
TTCN_ASP_Constraint (p. 1038) TTCN_PDU_Constraint (p. 1040)
ASN1_ASP_Constraint (p. 1010) ASN1_PDU_Constraint (p. 1012)
TTCN_CM_Constraint (p. 1039) ASN1_CM_Constraint (p. 1012)

ConsIdAndParList ::= SEQUENCE {

constraintIdentifier	Identifier		--See page 1100
formalParList	FormalParList		--See page 1060

}

Referenced from: ConsId (p. 1050)

ConsRef ::= SEQUENCE {

constraintIdentifier	Identifier		--See page 1100
actualCrefParList	ActualCrefParList		--See page 1041

}

Referenced from: ConstraintReference (p. 1051) ConstraintValue (p. 1051) Primary (p. 1074)

Constraint ::= SEQUENCE {

valueConstraint	ValueConstraint	OPTIONAL	--See page 1093
presenceConstraint	PresenceConstraint	OPTIONAL	--See page 1073

}

Referenced from: NamedConstraint (p. 1066)

ConstraintExpression ::= SEQUENCE {

expression	Expression	--See page 1060

}

Referenced from: ConstraintValue (p. 1051) LowerRangeBound (p. 1064) UpperRangeBound (p. 1091)

ConstraintReference ::= CHOICE {

consRef	ConsRef	--See page 1051
formalParIdentifier	Identifier	--See page 1100

}

Referenced from: Cref (p. 1054)

ConstraintValue ::= CHOICE {

constraintExpression	ConstraintExpression	--See page 1051
matchingSymbol	MatchingSymbol	--See page 1065
consRef	ConsRef	--See page 1051

}

Referenced from: ConstraintValueAndAttributes (p. 1052)

ConstraintValueAndAttributes ::= SEQUENCE {

constraintValue	ConstraintValue		--See page 1051
valueAttributes	ValueAttributes	OPTIONAL	--See page 1092

}

Referenced from: ConsValue (p. 1052) ConstraintValueAndAttributesOrReplace (p. 1052) Replace (p. 1076) ActualCrefParList (p. 1041) ValueList (p. 1093) SuperSet (p. 1083) SubSet (p. 1082) ActualParList (p. 1041) SingleValue (p. 1081) UpperEndValue (p. 1091) LowerEndValue (p. 1064) ASN1_ValueList (p. 1044) AnyValue (p. 1042) NamedValue (p. 1067) DefaultValue (p. 1055)

ConstraintValueAndAttributesOrReplace ::= CHOICE {

constraintValueAndAttributes	ConstraintValueAndAttributes	--See page 1052
replacementList	ReplacementList	--See page 1076

}

Referenced from: ASN1_ConsValue (p. 1043)

Construct ::= CHOICE {

goTo	GoTo	--See page 1061
attach	Attach	--See page 1045
repeat	Repeat	--See page 1076
return_	RETURN_	--See page 1106
activate	Activate	--See page 1041
create	Create	--See page 1053

}

Referenced from: StatementLine (p. 1081)

ConsValue ::= SEQUENCE {

constraintValueAndAttributes	ConstraintValueAndAttributes	--See page 1052

}

Referenced from: ElemValue (p. 1022) ASP_ParValue (p. 1016) PDU_FieldValue (p. 1026) CM_ParValue (p. 1018)

ContainedSubType ::= SEQUENCE {

asn1_Type	ASN1_Type	--See page 1043

}

Referenced from: SubtypeValueSet (p. 1082)

CP_Id ::= SEQUENCE {

cp_Identifier	Identifier	--See page 1100

}

Referenced from: CP_Dcl (p. 1019)

CP_List ::= SEQUENCE OF {

cp_Identifier	Identifier	--See page 1100

}

Referenced from: CPs_Used (p. 1053)

CPs_Used ::= SEQUENCE {

cp_List	CP_List	--See page 1053

}

Referenced from: TCompConfigInfo (p. 1031)

Create ::= SEQUENCE {

createList	CreateList	--See page 1053

}

Referenced from: Construct (p. 1052)

CreateAttach ::= SEQUENCE {

treeReference	TreeReference	--See page 1087
actualParList	ActualParList	--See page 1041

}

Referenced from: CreateTComp (p. 1053)

CreateList ::= SEQUENCE OF {

createTComp	CreateTComp	--See page 1053

}

Referenced from: Create (p. 1053)

CreateTComp ::= SEQUENCE {

tcompIdentifier	Identifier	--See page 1100
createAttach	CreateAttach	--See page 1053

}

Referenced from: CreateList (p. 1053)

Cref ::= SEQUENCE {
    constraintReference   ConstraintReference   --See page 1051
}
Referenced from: BehaviourLine (p. 1017)

DataObjectReference ::= SEQUENCE {
    dataObjectIdentifier   Identifier   --See page 1100
    componentReferenceList   ComponentReferenceList   --See page 1050
}
Referenced from: Assignment (p. 1045) ReadTimer (p. 1075) Primary (p. 1074)

DeclarationValue ::= SEQUENCE {
    expression   Expression   --See page 1060
}
Referenced from: TS_ConstValue (p. 1087) TS_VarValue (p. 1089) TC_VarValue (p. 1084) Duration (p. 1056)

DefaultExpression ::= SEQUENCE {
    expression   Expression   --See page 1060
}
Referenced from: EncodingDefault (p. 1057)

DefaultGroupId ::= SEQUENCE {
    defaultGroupIdentifier   Identifier   --See page 1100
}
Referenced from: DefaultGroup (p. 1021)

DefaultId ::= SEQUENCE {
    defaultIdAndParList   DefaultIdAndParList   --See page 1054
}
Referenced from: DefaultCase (p. 1020)

DefaultIdAndParList ::= SEQUENCE {
    defaultIdentifier   Identifier   --See page 1100
    formalParList   FormalParList   --See page 1060
}
Referenced from: DefaultId (p. 1054)

DefaultRef ::= SEQUENCE {
    defaultGroupReference DefaultGroupReference --See page 1097
}
Referenced from: DefaultCase (p. 1020)

DefaultReference ::= SEQUENCE {
    defaultIdentifier Identifier --See page 1100
    actualParList ActualParList --See page 1041
}
Referenced from: DefaultRefList (p. 1055)

DefaultRefList ::= SEQUENCE OF {
    defaultReference DefaultReference --See page 1055
}
Referenced from: DefaultsRef (p. 1055) Activate (p. 1041)

DefaultsRef ::= SEQUENCE {
    defaultRefList DefaultRefList --See page 1055
}
Referenced from: CompactTestGroup (p. 1019) TestCase (p. 1032) TestStep (p. 1033)

DefaultValue ::= SEQUENCE {
    constraintValueAndAttributes ConstraintValueAndAttributes --See page 1052
}
Referenced from: NamedTypeAttribute (p. 1067)

DefinedType ::= SEQUENCE {
    identifier Identifier --See page 1100
}
Referenced from: ASN1_Type (p. 1043)

DefinedValue ::= SEQUENCE {
    identifier Identifier --See page 1100
}
Referenced from: ClassNumber (p. 1048)

DerivationPath ::= SEQUENCE OF {
    constraintIdentifier Identifier --See page 1100
}
Referenced from: DerivPath (p. 1056)

DerivPath ::= SEQUENCE {
    derivationPath DerivationPath --See page 1056
}
Referenced from: StructTypeConstraint (p. 1029) ASN1_TypeConstraint (p. 1014) TTCN_ASP_Constraint (p. 1038) TTCN_PDU_Constraint (p. 1040) ASN1_ASP_Constraint (p. 1010) ASN1_PDU_Constraint (p. 1012) TTCN_CM_Constraint (p. 1039) ASN1_CM_Constraint (p. 1012)

Done ::= SEQUENCE {
    tcompIdList TCompIdList --See page 1084
}
Referenced from: Event (p. 1059)

Duration ::= SEQUENCE {
    declarationValue DeclarationValue --See page 1054
}
Referenced from: TimerDcl (p. 1034)

ElementType ::= CHOICE {
    namedTypeAndAttributes NamedTypeAndAttributes --See page 1067
    componentsOf ComponentsOf --See page 1050
}
Referenced from: ElementTypeList (p. 1056)

ElementTypeList ::= SEQUENCE OF {
    elementType ElementType --See page 1056
}
Referenced from: SequenceType (p. 1078) SetType (p. 1079)

ElementValueList ::= SEQUENCE OF {
    namedValue NamedValue --See page 1067
}
Referenced from: SequenceValue (p. 1078) SetValue (p. 1079)

ElemId ::= SEQUENCE {
    elemIdAndFullId   ElemIdAndFullId   --See page 1057
}
Referenced from: ElemDcl (p. 1022) ElemValue (p. 1022)

ElemIdAndFullId ::= SEQUENCE {
    elemIdentifier   Identifier   --See page 1100
    fullIdentifier   FullIdentifier   OPTIONAL   --See page 1099
}
Referenced from: ElemId (p. 1057)

ElemType ::= SEQUENCE {
    typeAndAttributes   TypeAndAttributes   --See page 1089
}
Referenced from: ElemDcl (p. 1022)

EncodingDefault ::= SEQUENCE {
    defaultExpression   DefaultExpression   --See page 1054
}
Referenced from: EncodingDefinition (p. 1022)

EncodingRef ::= SEQUENCE {
    encodingReference   EncodingReference   --See page 1098
}
Referenced from: EncodingDefinition (p. 1022)

EncodingRuleId ::= SEQUENCE {
    encodingRuleIdentifier   Identifier   --See page 1100
}
Referenced from: EncodingDefinition (p. 1022) EncodingVariationSet (p. 1024)

EncodingVariationId ::= SEQUENCE {
    encVariationIdAndParList   EncVariationIdAndParList   --See page 1058
}
Referenced from: EncodingVariation (p. 1023)

EncRuleId ::= SEQUENCE {
    encodingRuleIdentifier Identifier --See page 1100
}
Referenced from: TTCN_PDU_Constraint (p. 1040) ASN1_PDU_Constraint (p. 1012)

EncVariationCall ::= SEQUENCE {
    encVariationIdentifier Identifier --See page 1100
    actualParList ActualParList --See page 1041
}
Referenced from: PDU_FieldEncodingCall (p. 1072) EncVariationId (p. 1058)

EncVariationId ::= SEQUENCE {
    encVariationCall EncVariationCall --See page 1058
}
Referenced from: StructTypeDef (p. 1030) ASN1_TypeDef (p. 1014) ASN1_TypeRef (p. 1015) TTCN_PDU_TypeDef (p. 1040) ASN1_PDU_TypeDef (p. 1013) ASN1_PDU_TypeDefByRef (p. 1013) StructTypeConstraint (p. 1029) ASN1_TypeConstraint (p. 1014) TTCN_PDU_Constraint (p. 1040) ASN1_PDU_Constraint (p. 1012)

EncVariationIdAndParList ::= SEQUENCE {
    encVariationIdentifier Identifier --See page 1100
    formalParList FormalParList --See page 1060
}
Referenced from: EncodingVariationId (p. 1057)

EnumeratedType ::= SEQUENCE {
    enumeration Enumeration --See page 1058
}
Referenced from: BuiltinType (p. 1047)

Enumeration ::= SEQUENCE OF {
    namedNumber NamedNumber --See page 1066
}
Referenced from: EnumeratedType (p. 1058)

Event ::= CHOICE {

send	Send		--See page 1078
receive	Receive		--See page 1075
otherwise	Otherwise		--See page 1069
timeout	Timeout		--See page 1086
done	Done		--See page 1056

}

Referenced from: EventStatement (p. 1059)

EventStatement ::= SEQUENCE {

event	Event	OPTIONAL	--See page 1059
qualifier	Qualifier	OPTIONAL	--See page 1074
assignmentList	AssignmentList		--See page 1045
timerOps	TimerOps		--See page 1086

}

Referenced from: StatementLine (p. 1081)

ExpandedId ::= SEQUENCE {

expansion	Expansion		--See page 1059

}

Referenced from: AliasDef (p. 1010)

Expansion ::= SEQUENCE {

identifier	Identifier		--See page 1100

}

Referenced from: ExpandedId (p. 1059)

Exponent ::= CHOICE {

signedNumber	SignedNumber		--See page 1079
number	Number		--See page 1103

}

Referenced from: NumericRealValue (p. 1068)

Expression ::= CHOICE {

relExpression	RelExpression	--See page 1076
simpleExpression	SimpleExpression	--See page 1079

}

Referenced from: VariationDefault (p. 1093) SelectionExpression (p. 1077) DeclarationValue (p. 1054) Assignment (p. 1045) Qualifier (p. 1074) TimerValue (p. 1086) ConstraintExpression (p. 1051) DefaultExpression (p. 1054) ParenExpression (p. 1070) ComponentNumber (p. 1050) BitNumber (p. 1046)

Factor ::= CHOICE {

unaryExpression	UnaryExpression	--See page 1090
primary	Primary	--See page 1074

}

Referenced from: Term (p. 1085) MultExpression (p. 1065)

Fail ::= CHOICE {

fail	FAIL	--See page 1098
preliminaryFAIL	PreliminaryFAIL	--See page 1105

}

Referenced from: Verdict (p. 1094)

FormalParAndType ::= SEQUENCE {

formalParIdList	FormalParIdList	--See page 1060
formalParType	FormalParType	--See page 1061

}

Referenced from: FormalParList (p. 1060)

FormalParIdList ::= SEQUENCE OF {

formalParIdentifier	Identifier	--See page 1100

}

Referenced from: FormalParAndType (p. 1060)

FormalParList ::= SEQUENCE OF {

formalParAndType	FormalParAndType	--See page 1060

}

Referenced from: EncVariationIdAndParList (p. 1058) InvalidFieldEncodingIdAndParList (p. 1063) TS_OpIdAndParList (p. 1088) TS_ProcIdAndParList (p. 1088) ConsIdAndParList (p. 1051) TreeHeader (p. 1087) TestStepIdAndParList (p. 1086) DefaultIdAndParList (p. 1054)

FormalParType ::= CHOICE {

ttcn_Type	TTCN_Type	--See page 1089
pdu	PDU	--See page 1104
cp	CP	--See page 1097
timer	TIMER	--See page 1108

}

Referenced from: FormalParAndType (p. 1060)

FullSpecification ::= SEQUENCE {

typeConstraints	TypeConstraints	--See page 1090

}

Referenced from: MultipleTypeConstraints (p. 1065)

GoTo ::= SEQUENCE {

label	Label	--See page 1063

}

Referenced from: Construct (p. 1052)

Header ::= SEQUENCE {

treeHeader	TreeHeader	--See page 1087

}

Referenced from: LocalTree (p. 1024)

IdentifierList ::= SEQUENCE OF {

identifier	Identifier	--See page 1100

}

Referenced from: BitStringValue (p. 1046)

IdOrNum ::= CHOICE {

identifier	Identifier	--See page 1100
number	Number	--See page 1103
signedNumber	SignedNumber	--See page 1079

}

Referenced from: NamedNumber (p. 1066)

ImplicitSend ::= SEQUENCE {

identifier	Identifier	--See page 1100

}

Referenced from: StatementLine (p. 1081)

Inconclusive ::= CHOICE {

inconc	INCONC	--See page 1101
preliminaryINCONC	PreliminaryINCONC	--See page 1105

}

Referenced from: Verdict (p. 1094)

Indentation ::= SEQUENCE {

number	Number	--See page 1103

}

Referenced from: Line (p. 1063)

InnerTypeConstraints ::= CHOICE {

withComponent	WithComponent	--See page 1094
withComponents	WithComponents	--See page 1094

}

Referenced from: SubtypeValueSet (p. 1082)

IntegerRange ::= SEQUENCE {

lowerTypeBound	LowerTypeBound	--See page 1064
upperTypeBound	UpperTypeBound	--See page 1092

}

Referenced from: Restriction (p. 1077)

IntegerType ::= SEQUENCE {

namedNumberList	NamedNumberList	--See page 1067

}

Referenced from: BuiltinType (p. 1047)

InvalidFieldEncodingCall ::= SEQUENCE {

invalidFieldEncodingIdentifier	Identifier	--See page 1100
actualParList	ActualParList	--See page 1041

}

Referenced from: PDU_FieldEncodingCall (p. 1072)

InvalidFieldEncodingId ::= SEQUENCE {

invalidFieldEncodingIdAndParList	InvalidFieldEncodingIdAndParList	--See page 1063

}

Referenced from: InvalidFieldEncodingDef (p. 1024)

InvalidFieldEncodingIdAndParList ::= SEQUENCE {

invalidFieldEncodingIdentifier	Identifier	--See page 1100
formalParList	FormalParList	--See page 1060

}

Referenced from: InvalidFieldEncodingId (p. 1062)

Label ::= SEQUENCE {

identifier	Identifier	--See page 1100

}

Referenced from: LabelId (p. 1063) GoTo (p. 1061)

LabelId ::= SEQUENCE {

label	Label	--See page 1063

}

Referenced from: BehaviourLine (p. 1017)

LengthAttribute ::= CHOICE {

singleLength	SingleLength	--See page 1080
rangeLength	RangeLength	--See page 1074

}

Referenced from: TypeAndLengthAttribute (p. 1090)

LengthRestriction ::= CHOICE {

singleTypeLength	SingleTypeLength	--See page 1080
rangeTypeLength	RangeTypeLength	--See page 1074

}

Referenced from: Restriction (p. 1077)

Line ::= SEQUENCE {

indentation	Indentation	--See page 1062
statementLine	StatementLine	--See page 1081

}

Referenced from: BehaviourLine (p. 1017)

LineNumber ::= SEQUENCE {

number	Number	--See page 1103

}

Referenced from: BehaviourLine (p. 1017)

LiteralValue ::= CHOICE {

number	Number		--See page 1103
booleanValue	BooleanValue		--See page 1046
bstring	Bstring		--See page 1096
hstring	Hstring		--See page 1099
ostring	Ostring		--See page 1104
cstring	Cstring		--See page 1097
r_Value	R_Value		--See page 1074

}

Referenced from: SimpleValue (p. 1080) Value (p. 1092)

LowerBound ::= SEQUENCE {

bound	Bound		--See page 1047

}

Referenced from: RangeLength (p. 1074)

LowerEndpoint ::= SEQUENCE {

lowerEndValue	LowerEndValue		--See page 1064
inclusive	INCLUSIVE	OPTIONAL	--See page 1101

}

Referenced from: ValueRange (p. 1093)

LowerEndValue ::= CHOICE {

constraintValueAndAttributes	ConstraintValueAndAttributes	--See page 1052
min	Min	--See page 1102

}

Referenced from: LowerEndpoint (p. 1064)

LowerRangeBound ::= CHOICE {

constraintExpression	ConstraintExpression	--See page 1051
minusINFINITY	MinusINFINITY	--See page 1102

}

Referenced from: ValRange (p. 1092)

LowerTypeBound ::= CHOICE {

signedNumber	SignedNumber	--See page 1079
number	Number	--See page 1103
minusINFINITY	MinusINFINITY	--See page 1102

}

Referenced from: RangeTypeLength (p. 1074) IntegerRange (p. 1062)

LowerValueBound ::= SEQUENCE {

valueBound	ValueBound	--See page 1093

}

Referenced from: RangeValueLength (p. 1074)

Mantissa ::= CHOICE {

signedNumber	SignedNumber	--See page 1079
number	Number	--See page 1103

}

Referenced from: NumericRealValue (p. 1068)

MatchingSymbol ::= CHOICE {

complement	Complement	--See page 1049
omit	Omit	--See page 1069
anyValue	ANYValue	--See page 1095
anyOrOmit	AnyOrOmit	--See page 1095
valueList	ValueList	--See page 1093
valRange	ValRange	--See page 1092
superSet	SuperSet	--See page 1083
subSet	SubSet	--See page 1082
permutation	Permutation	--See page 1073

}

Referenced from: ConstraintValue (p. 1051)

MultExpression ::= SEQUENCE {

term	Term	--See page 1085
multOp	MultOp	--See page 1066
factor	Factor	--See page 1060

}

Referenced from: Term (p. 1085)

MultipleTypeConstraints ::= CHOICE {

fullSpecification	FullSpecification	--See page 1061
partialSpecification	PartialSpecification	--See page 1070

}

Referenced from: WithComponents (p. 1094)

MultOp ::= CHOICE {

mult	Mult	--See page 1103
div	Div	--See page 1098
mod	Mod	--See page 1102
and	And	--See page 1095

}

Referenced from: MultExpression (p. 1065)

NameAndNumberForm ::= SEQUENCE {

nameForm	NameForm	--See page 1067
numberForm	NumberForm	--See page 1068

}

Referenced from: ObjIdComponent (p. 1069)

NamedBitList ::= SEQUENCE OF {

namedNumber	NamedNumber	--See page 1066

}

Referenced from: BitStringType (p. 1046)

NamedConstraint ::= SEQUENCE {

identifier	Identifier	--See page 1100
constraint	Constraint	--See page 1051

}

Referenced from: NamedConstraintList (p. 1066)

NamedConstraintList ::= SEQUENCE OF {

namedConstraint	NamedConstraint	--See page 1066

}

Referenced from: TypeConstraints (p. 1090)

NamedNumber ::= SEQUENCE {

identifier	Identifier	--See page 1100
idOrNum	IdOrNum	--See page 1061

}

Referenced from: NamedNumberList (p. 1067) Enumeration (p. 1058) NamedBitList (p. 1066)

NamedNumberList ::= SEQUENCE OF {
    namedNumber   NamedNumber   --See page 1066
}
Referenced from: IntegerType (p. 1062)

NamedType ::= SEQUENCE {
    identifier   Identifier   --See page 1100
    asn1_Type   ASN1_Type   --See page 1043
}
Referenced from: AlternativeTypeList (p. 1042) NamedTypeOrSelection (p. 1067)

NamedTypeAndAttributes ::= SEQUENCE {
    namedTypeOrSelection   NamedTypeOrSelection   --See page 1067
    namedTypeAttribute   NamedTypeAttribute   OPTIONAL   --See page 1067
}
Referenced from: ElementType (p. 1056)

NamedTypeAttribute ::= CHOICE {
    optional   OPTIONAL   --See page 1104
    defaultValue   DefaultValue   --See page 1055
}
Referenced from: NamedTypeAndAttributes (p. 1067)

NamedTypeOrSelection ::= CHOICE {
    namedType   NamedType   --See page 1067
    selectionType   SelectionType   --See page 1077
}
Referenced from: NamedTypeAndAttributes (p. 1067)

NamedValue ::= SEQUENCE {
    identifier   Identifier   --See page 1100
    constraintValueAndAttributes   ConstraintValueAndAttributes   --See page 1052
}
Referenced from: ElementValueList (p. 1056) ChoiceValue (p. 1048) SelectionValue (p. 1077)

NameForm ::= SEQUENCE {
    identifier   Identifier   --See page 1100
}
Referenced from: ObjIdComponent (p. 1069) NameAndNumberForm (p. 1066)

Num_CPs ::= SEQUENCE {
    number Number --See page 1103
}
Referenced from: NumOf_CPs (p. 1068)

Num_PCOs ::= SEQUENCE {
    number Number --See page 1103
}
Referenced from: NumOf_PCOs (p. 1068)

NumberForm ::= SEQUENCE {
    number Number --See page 1103
}
Referenced from: ObjIdComponent (p. 1069) NameAndNumberForm (p. 1066)

NumericRealValue ::= SEQUENCE {
    mantissa Mantissa --See page 1065
    base Base --See page 1046
    exponent Exponent --See page 1059
}
Referenced from: RealValue (p. 1075)

NumOf_CPs ::= SEQUENCE {
    num_CPs Num_CPs --See page 1068
}
Referenced from: TCompDcl (p. 1032)

NumOf_PCOs ::= SEQUENCE {
    num_PCOs Num_PCOs --See page 1068
}
Referenced from: TCompDcl (p. 1032)

ObjectIdentifierValue ::= SEQUENCE {
    objIdComponentList ObjIdComponentList --See page 1069
}
Referenced from: ASN1_Value (p. 1044)

ObjIdComponent ::= CHOICE {

nameForm	NameForm		--See page 1067
numberForm	NumberForm		--See page 1068
nameAndNumberForm	NameAndNumberForm		--See page 1066

}

Referenced from: ObjIdComponentList (p. 1069)

ObjIdComponentList ::= SEQUENCE OF {

objIdComponent	ObjIdComponent		--See page 1069

}

Referenced from: ObjectIdentifierValue (p. 1068)

Omit ::= SEQUENCE {

omit	OMIT		--See page 1104

}

Referenced from: MatchingSymbol (p. 1065)

OmitReference ::= SEQUENCE {

referenceList	ReferenceList		--See page 1075

}

Referenced from: Replacement (p. 1076)

OpCall ::= SEQUENCE {

ts_OpIdentifier	Identifier		--See page 1100
actualParList	ActualParList		--See page 1041

}

Referenced from: Primary (p. 1074)

Otherwise ::= SEQUENCE {

identifier	Identifier	OPTIONAL	--See page 1100

}

Referenced from: Event (p. 1059)

P_Role ::= SEQUENCE {

pco_Role	PCO_Role		--See page 1071

}

Referenced from: PCO_TypeDcl (p. 1025) PCO_Dcl (p. 1025)

ParenExpression ::= SEQUENCE {
    expression Expression --See page 1060
}
Referenced from: Primary (p. 1074)

ParentSubType ::= SEQUENCE {
    parentType ParentType --See page 1070
    subTypeSpec SubTypeSpec --See page 1082
}
Referenced from: SubType (p. 1082)

ParentType ::= SEQUENCE {
    asn1_Type ASN1_Type --See page 1043
}
Referenced from: ParentSubType (p. 1070)

PartialSpecification ::= SEQUENCE {
    typeConstraints TypeConstraints --See page 1090
}
Referenced from: MultipleTypeConstraints (p. 1065)

Pass ::= CHOICE {
    pass PASS --See page 1104
    preliminaryPASS PreliminaryPASS --See page 1105
}
Referenced from: Verdict (p. 1094)

PCO_Id ::= SEQUENCE {
    pco_Identifier Identifier --See page 1100
}
Referenced from: PCO_Dcl (p. 1025)

PCO_List ::= SEQUENCE OF {
    pco_Identifier Identifier --See page 1100
}
Referenced from: PCOs_Used (p. 1071)

PCO_Role ::= CHOICE {
    ut UT --See page 1108
    lt LT --See page 1101
}
Referenced from: P_Role (p. 1069)

PCO_Type ::= SEQUENCE {
    pco_TypeIdentifier Identifier --See page 1100
}
Referenced from: TTCN_ASP_TypeDef (p. 1039) ASN1_ASP_TypeDef (p. 1011) ASN1_ASP_TypeDefByRef (p. 1011) TTCN_PDU_TypeDef (p. 1040) ASN1_PDU_TypeDef (p. 1013) ASN1_PDU_TypeDefByRef (p. 1013)

PCO_TypeId ::= SEQUENCE {
    pco_TypeIdentifier Identifier --See page 1100
}
Referenced from: PCO_TypeDcl (p. 1025) PCO_Dcl (p. 1025)

PCOs_Used ::= SEQUENCE {
    pco_List PCO_List --See page 1070
}
Referenced from: TCompConfigInfo (p. 1031)

PDU_EncodingId ::= SEQUENCE {
    encodingRuleIdentifier Identifier --See page 1100
}
Referenced from: TTCN_PDU_TypeDef (p. 1040) ASN1_PDU_TypeDef (p. 1013) ASN1_PDU_TypeDefByRef (p. 1013)

PDU_FieldEncoding ::= SEQUENCE {
    pdu_FieldEncodingCall PDU_FieldEncodingCall --See page 1072
}
Referenced from: SimpleTypeDef (p. 1027) ElemDcl (p. 1022) PDU_FieldDcl (p. 1026) PDU_FieldValue (p. 1026)

PDU_FieldEncodingCall ::= CHOICE {

encVariationCall	EncVariationCall		--See page 1058
invalidFieldEncodingCall	InvalidFieldEncodingCall		--See page 1062

}

Referenced from: PDU_FieldEncoding (p. 1071)

PDU_FieldId ::= SEQUENCE {

pdu_FieldIdOrMacro	PDU_FieldIdOrMacro		--See page 1072

}

Referenced from: PDU_FieldDcl (p. 1026) PDU_FieldValue (p. 1026)

PDU_FieldIdAndFullId ::= SEQUENCE {

pdu_FieldIdentifier	Identifier		--See page 1100
fullIdentifier	FullIdentifier	OPTIONAL	--See page 1099

}

Referenced from: PDU_FieldIdOrMacro (p. 1072)

PDU_FieldIdOrMacro ::= CHOICE {

pdu_FieldIdAndFullId	PDU_FieldIdAndFullId		--See page 1072
macroSymbol	MacroSymbol		--See page 1102

}

Referenced from: PDU_FieldId (p. 1072)

PDU_FieldType ::= SEQUENCE {

typeAndAttributes	TypeAndAttributes		--See page 1089

}

Referenced from: PDU_FieldDcl (p. 1026)

PDU_Id ::= SEQUENCE {

pdu_IdAndFullId	PDU_IdAndFullId		--See page 1073

}

Referenced from: TTCN_PDU_TypeDef (p. 1040) ASN1_PDU_TypeDef (p. 1013) ASN1_PDU_TypeDefByRef (p. 1013) TTCN_PDU_Constraint (p. 1040) ASN1_PDU_Constraint (p. 1012)

PDU_IdAndFullId ::= SEQUENCE {

pdu_Identifier	Identifier		--See page 1100
fullIdentifier	FullIdentifier	OPTIONAL	--See page 1099

}

Referenced from: PDU_Id (p. 1072)

PermittedAlphabet ::= SEQUENCE {

subTypeSpec	SubTypeSpec	--See page 1082

}

Referenced from: SubtypeValueSet (p. 1082)

Permutation ::= SEQUENCE {

valueList	ValueList	--See page 1093

}

Referenced from: MatchingSymbol (p. 1065)

PredefinedType ::= CHOICE {

integer	INTEGER	--See page 1101
boolean	BOOLEAN	--See page 1096
bitstring	BITSTRING	--See page 1096
hexstring	HEXSTRING	--See page 1099
octetstring	OCTETSTRING	--See page 1104
r_Type	R_TYPE	--See page 1106
characterString	CharacterString	--See page 1048

}

Referenced from: TTCN_Type (p. 1089)

PresenceConstraint ::= CHOICE {

present	PRESENT	--See page 1105
absent	ABSENT	--See page 1095
optional	OPTIONAL	--See page 1104

}

Referenced from: Constraint (p. 1051)

Primary ::= CHOICE {

value	Value	--See page 1092
dataObjectReference	DataObjectReference	--See page 1054
opCall	OpCall	--See page 1069
consRef	ConsRef	--See page 1051
selectExprId	SelectExprId	--See page 1077
parenExpression	ParenExpression	--See page 1070

}

Referenced from: Factor (p. 1060) UnaryExpression (p. 1090)

Qualifier ::= SEQUENCE {

expression	Expression	--See page 1060

}

Referenced from: EventStatement (p. 1059) Repeat (p. 1076)

R_Value ::= CHOICE {

pass	PASS	--See page 1104
fail	FAIL	--See page 1098
inconc	INCONC	--See page 1101
none	NONE	--See page 1103

}

Referenced from: LiteralValue (p. 1064)

RangeLength ::= SEQUENCE {

lowerBound	LowerBound	--See page 1064
upperBound	UpperBound	--See page 1091

}

Referenced from: LengthAttribute (p. 1063)

RangeTypeLength ::= SEQUENCE {

lowerTypeBound	LowerTypeBound	--See page 1064
upperTypeBound	UpperTypeBound	--See page 1092

}

Referenced from: LengthRestriction (p. 1063)

RangeValueLength ::= SEQUENCE {

lowerValueBound	LowerValueBound	--See page 1065
upperValueBound	UpperValueBound	--See page 1092

}

Referenced from: ValueLength (p. 1093)

ReadTimer ::= SEQUENCE {

timerIdentifier	Identifier		--See page 1100
dataObjectReference	DataObjectReference		--See page 1054

}

Referenced from: TimerOp (p. 1086)

RealValue ::= CHOICE {

numericRealValue	NumericRealValue		--See page 1068
specialRealValue	SpecialRealValue		--See page 1081

}

Referenced from: ASN1_Value (p. 1044)

Receive ::= SEQUENCE {

identifier1	Identifier	OPTIONAL	--See page 1100
identifier2	Identifier		--See page 1100

}

Referenced from: Event (p. 1059)

RecordRef ::= SEQUENCE {

componentIdentifier	ComponentIdentifier		--See page 1049

}

Referenced from: ComponentReference (p. 1050)

ReferenceList ::= SEQUENCE {

arrayRefOrComp	ArrayRefOrComp		--See page 1042
componentReferenceList	ComponentReferenceList		--See page 1050

}

Referenced from: Replace (p. 1076) OmitReference (p. 1069)

ReferenceType ::= SEQUENCE {

identifier	Identifier		--See page 1100

}

Referenced from: TTCN_Type (p. 1089) AnyValue (p. 1042)

RelExpression ::= SEQUENCE {

simpleExpression1	SimpleExpression	--See page 1079
relOp	RelOp	--See page 1076
simpleExpression2	SimpleExpression	--See page 1079

}

Referenced from: Expression (p. 1060)

RelOp ::= CHOICE {

equal	Equal	--See page 1098
less	Less	--See page 1101
greater	Greater	--See page 1099
notEqual	NotEqual	--See page 1103
greaterOrEqual	GreaterOrEqual	--See page 1099
lessOrEqual	LessOrEqual	--See page 1101

}

Referenced from: RelExpression (p. 1076)

Repeat ::= SEQUENCE {

attach	Attach	--See page 1045
qualifier	Qualifier	--See page 1074

}

Referenced from: Construct (p. 1052)

Replace ::= SEQUENCE {

referenceList	ReferenceList	--See page 1075
constraintValueAndAttributes	ConstraintValueAndAttributes	--See page 1052

}

Referenced from: Replacement (p. 1076)

Replacement ::= CHOICE {

replace	Replace	--See page 1076
omitReference	OmitReference	--See page 1069

}

Referenced from: ReplacementList (p. 1076)

ReplacementList ::= SEQUENCE OF {

replacement	Replacement	--See page 1076

}

Referenced from: ConstraintValueAndAttributesOrReplace (p. 1052)

Restriction ::= CHOICE {

lengthRestriction	LengthRestriction	--See page 1063
integerRange	IntegerRange	--See page 1062
simpleValueList	SimpleValueList	--See page 1080

}

Referenced from: TypeAndRestriction (p. 1090)

Result ::= SEQUENCE {

r	R	--See page 1105

}

Referenced from: Verdict (p. 1094)

SelectExpr ::= SEQUENCE {

selectionExpression	SelectionExpression	--See page 1077

}

Referenced from: SelectExprDef (p. 1027)

SelectExprId ::= SEQUENCE {

selectExprIdentifier	Identifier	--See page 1100

}

Referenced from: SelectExprDef (p. 1027) Primary (p. 1074)

SelectionExpression ::= SEQUENCE {

expression	Expression	--See page 1060

}

Referenced from: SelectExpr (p. 1077)

SelectionType ::= SEQUENCE {

identifier	Identifier	--See page 1100
asn1_Type	ASN1_Type	--See page 1043

}

Referenced from: BuiltinType (p. 1047) NamedTypeOrSelection (p. 1067)

SelectionValue ::= SEQUENCE {

namedValue	NamedValue	--See page 1067

}

Referenced from: ASN1_Value (p. 1044)

SelExprId ::= SEQUENCE {
    selectExprIdentifier Identifier --See page 1100
}
Referenced from: TestGroup (p. 1033) CompactTestGroup (p. 1019) CompactTestCase (p. 1018) TestCase (p. 1032)

Send ::= SEQUENCE {
    identifier1 Identifier OPTIONAL --See page 1100
    identifier2 Identifier --See page 1100
}
Referenced from: Event (p. 1059)

SequenceOfType ::= SEQUENCE {
    asn1_Type ASN1_Type OPTIONAL --See page 1043
}
Referenced from: BuiltinType (p. 1047)

SequenceOfValue ::= SEQUENCE {
    asn1_ValueList ASN1_ValueList --See page 1044
}
Referenced from: ASN1_Value (p. 1044)

SequenceSubType ::= SEQUENCE {
    sizeConstraint SizeConstraint --See page 1081
    asn1_Type ASN1_Type --See page 1043
}
Referenced from: SubType (p. 1082)

SequenceType ::= SEQUENCE {
    elementTypeList ElementTypeList --See page 1056
}
Referenced from: BuiltinType (p. 1047)

SequenceValue ::= SEQUENCE {
    elementValueList ElementValueList --See page 1056
}
Referenced from: ASN1_Value (p. 1044)

SetOfType ::= SEQUENCE {

asn1_Type	ASN1_Type	OPTIONAL	--See page 1043

}

Referenced from: BuiltinType (p. 1047)

SetOfValue ::= SEQUENCE {

asn1_ValueList	ASN1_ValueList	--See page 1044

}

Referenced from: ASN1_Value (p. 1044)

SetSubType ::= SEQUENCE {

sizeConstraint	SizeConstraint	--See page 1081
asn1_Type	ASN1_Type	--See page 1043

}

Referenced from: SubType (p. 1082)

SetType ::= SEQUENCE {

elementTypeList	ElementTypeList	--See page 1056

}

Referenced from: BuiltinType (p. 1047)

SetValue ::= SEQUENCE {

elementValueList	ElementValueList	--See page 1056

}

Referenced from: ASN1_Value (p. 1044)

SignedNumber ::= SEQUENCE {

minus	Minus	--See page 1102
number	Number	--See page 1103

}

Referenced from: LowerTypeBound (p. 1064) UpperTypeBound (p. 1092) SimpleValue (p. 1080) IdOrNum (p. 1061) Mantissa (p. 1065) Exponent (p. 1059)

SimpleExpression ::= CHOICE {

addExpression	AddExpression	--See page 1041
term	Term	--See page 1085

}

Referenced from: Expression (p. 1060) RelExpression (p. 1076) RelExpression (p. 1076) AddExpression (p. 1041)

SimpleTypeDefinition ::= SEQUENCE {
    typeAndRestriction TypeAndRestriction --See page 1090
}
Referenced from: SimpleTypeDef (p. 1027)

SimpleTypeId ::= SEQUENCE {
    simpleTypeIdentifier Identifier --See page 1100
}
Referenced from: SimpleTypeDef (p. 1027)

SimpleValue ::= CHOICE {
    literalValue LiteralValue --See page 1064
    signedNumber SignedNumber --See page 1079
}
Referenced from: SimpleValueList (p. 1080)

SimpleValueList ::= SEQUENCE OF {
    simpleValue SimpleValue --See page 1080
}
Referenced from: Restriction (p. 1077)

SingleLength ::= SEQUENCE {
    bound Bound --See page 1047
}
Referenced from: LengthAttribute (p. 1063)

SingleTypeConstraint ::= SEQUENCE {
    subTypeSpec SubTypeSpec --See page 1082
}
Referenced from: WithComponent (p. 1094)

SingleTypeLength ::= SEQUENCE {
    number Number --See page 1103
}
Referenced from: LengthRestriction (p. 1063)

SingleValue ::= SEQUENCE {

constraintValueAndAttributes	ConstraintValueAndAttributes		--See page 1052

}

Referenced from: SubtypeValueSet (p. 1082)

SingleValueLength ::= SEQUENCE {

valueBound	ValueBound		--See page 1093

}

Referenced from: ValueLength (p. 1093)

SizeConstraint ::= SEQUENCE {

subTypeSpec	SubTypeSpec		--See page 1082

}

Referenced from: SetSubType (p. 1079) SequenceSubType (p. 1078) SubtypeValueSet (p. 1082)

SO_SuiteId ::= SEQUENCE {

suiteIdentifier	Identifier		--See page 1100

}

Referenced from: SuiteStructure (p. 1030)

SpecialRealValue ::= CHOICE {

plus_INFINITY	Plus_INFINITY		--See page 1105
minus_INFINITY	Minus_INFINITY		--See page 1102

}

Referenced from: RealValue (p. 1075)

StartTimer ::= SEQUENCE {

timerIdentifier	Identifier		--See page 1100
timerValue	TimerValue	OPTIONAL	--See page 1086

}

Referenced from: TimerOp (p. 1086)

StatementLine ::= CHOICE {

eventStatement	EventStatement		--See page 1059
construct	Construct		--See page 1052
implicitSend	ImplicitSend		--See page 1061

}

Referenced from: Line (p. 1063)

StructId ::= SEQUENCE {

structIdAndFullId	StructIdAndFullId		--See page 1082

}

Referenced from: StructTypeDef (p. 1030) StructTypeConstraint (p. 1029)

StructIdAndFullId ::= SEQUENCE {

structIdentifier	Identifier		--See page 1100
fullIdentifier	FullIdentifier	OPTIONAL	--See page 1099

}

Referenced from: StructId (p. 1082)

SubSet ::= SEQUENCE {

constraintValueAndAttributes	ConstraintValueAndAttributes	--See page 1052

}

Referenced from: MatchingSymbol (p. 1065)

SubType ::= CHOICE {

parentSubType	ParentSubType	--See page 1070
setSubType	SetSubType	--See page 1079
sequenceSubType	SequenceSubType	--See page 1078

}

Referenced from: ASN1_Type (p. 1043)

SubTypeSpec ::= SEQUENCE OF {

subtypeValueSet	SubtypeValueSet	--See page 1082

}

Referenced from: PermittedAlphabet (p. 1073) SizeConstraint (p. 1081) ParentSubType (p. 1070) SingleTypeConstraint (p. 1080) ValueConstraint (p. 1093)

SubtypeValueSet ::= CHOICE {

singleValue	SingleValue	--See page 1081
containedSubType	ContainedSubType	--See page 1052
valueRange	ValueRange	--See page 1093
permittedAlphabet	PermittedAlphabet	--See page 1073
sizeConstraint	SizeConstraint	--See page 1081
innerTypeConstraints	InnerTypeConstraints	--See page 1062

}

Referenced from: SubTypeSpec (p. 1082)

SuiteId ::= SEQUENCE {

suiteIdentifier	Identifier		--See page 1100

}

Referenced from: ASuite (p. 1016)

SuperSet ::= SEQUENCE {

constraintValueAndAttributes	ConstraintValueAndAttributes		--See page 1052

}

Referenced from: MatchingSymbol (p. 1065)

Tag ::= SEQUENCE {

classType	ClassType	OPTIONAL	--See page 1048
classNumber	ClassNumber		--See page 1048

}

Referenced from: TaggedType (p. 1083)

TagAttributes ::= CHOICE {

Aimplicit	AIMPLICIT	--See page 1095
Aexplicit	AEXPLICIT	--See page 1095

}

Referenced from: TaggedType (p. 1083)

TaggedType ::= SEQUENCE {

tag	Tag		--See page 1083
tagAttributes	TagAttributes	OPTIONAL	--See page 1083
asn1_Type	ASN1_Type		--See page 1043

}

Referenced from: BuiltinType (p. 1047)

TC_Role ::= SEQUENCE {

tcomp_Role	TComp_Role		--See page 1084

}

Referenced from: TCompDcl (p. 1032)

TC_VarId ::= SEQUENCE {

tc_VarIdentifier	Identifier		--See page 1100

}

Referenced from: TC_VarDcl (p. 1030)

TC_VarType ::= SEQUENCE {
    ttcn_Type TTCN_Type --See page 1089
}
Referenced from: TC_VarDcl (p. 1030)

TC_VarValue ::= SEQUENCE {
    declarationValue DeclarationValue --See page 1054
}
Referenced from: TC_VarDcl (p. 1030)

TComp_Role ::= CHOICE {
    mtc MTC --See page 1102
    ptc PTC --See page 1105
}
Referenced from: TC_Role (p. 1083)

TCompConfigId ::= SEQUENCE {
    tcompConfigIdentifier Identifier --See page 1100
}
Referenced from: TCompConfigDcl (p. 1031)

TCompId ::= SEQUENCE {
    tcompIdentifier Identifier --See page 1100
}
Referenced from: TCompDcl (p. 1032)

TCompIdList ::= SEQUENCE OF {
    tCompIdentifier Identifier --See page 1100
}
Referenced from: Done (p. 1056)

TCompUsed ::= SEQUENCE {
    tcompIdentifier Identifier --See page 1100
}
Referenced from: TCompConfigInfo (p. 1031)

Term ::= CHOICE {
    multExpression MultExpression --See page 1065
    factor Factor --See page 1060
}
Referenced from: SimpleExpression (p. 1079) AddExpression (p. 1041) MultExpression (p. 1065)

TestCaseId ::= SEQUENCE {
    testCaseIdentifier Identifier --See page 1100
}
Referenced from: CompactTestCase (p. 1018) TestCase (p. 1032)

TestGroupId ::= SEQUENCE {
    testGroupIdentifier Identifier --See page 1100
}
Referenced from: TestGroup (p. 1033) CompactTestGroup (p. 1019)

TestGroupRef ::= SEQUENCE {
    testGroupReference TestGroupReference --See page 1107
}
Referenced from: TestCase (p. 1032)

TestStepAttachment ::= SEQUENCE {
    attach Attach --See page 1045
}
Referenced from: CompactTestCase (p. 1018)

TestStepGroupId ::= SEQUENCE {
    testStepGroupIdentifier Identifier --See page 1100
}
Referenced from: TestStepGroup (p. 1034)

TestStepId ::= SEQUENCE {
    testStepIdAndParList TestStepIdAndParList --See page 1086
}
Referenced from: TestStep (p. 1033)

TestStepIdAndParList ::= SEQUENCE {

testStepIdentifier	Identifier		--See page 1100
formalParList	FormalParList		--See page 1060

}

Referenced from: TestStepId (p. 1085)

TestStepRef ::= SEQUENCE {

testStepGroupReference	TestStepGroupReference		--See page 1108

}

Referenced from: TestStep (p. 1033)

Timeout ::= SEQUENCE {

timerIdentifier	Identifier	OPTIONAL	--See page 1100

}

Referenced from: Event (p. 1059)

TimerId ::= SEQUENCE {

timerIdentifier	Identifier	--See page 1100

}

Referenced from: TimerDcl (p. 1034)

TimerOp ::= CHOICE {

startTimer	StartTimer	--See page 1081
cancelTimer	CancelTimer	--See page 1047
readTimer	ReadTimer	--See page 1075

}

Referenced from: TimerOps (p. 1086)

TimerOps ::= SEQUENCE OF {

timerOp	TimerOp	--See page 1086

}

Referenced from: EventStatement (p. 1059)

TimerValue ::= SEQUENCE {

expression	Expression	--See page 1060

}

Referenced from: StartTimer (p. 1081)

TimeUnit ::= CHOICE {

picoSeconds	PicoSeconds	--See page 1104
nanoSeconds	NanoSeconds	--See page 1103
microSeconds	MicroSeconds	--See page 1102
milliSeconds	MilliSeconds	--See page 1102
seconds	Seconds	--See page 1106
minutes	Minutes	--See page 1102

}

Referenced from: Unit (p. 1091)

TreeHeader ::= SEQUENCE {

treeIdentifier	Identifier	--See page 1100
formalParList	FormalParList	--See page 1060

}

Referenced from: Header (p. 1061)

TreeReference ::= SEQUENCE {

identifier	Identifier	--See page 1100

}

Referenced from: Attach (p. 1045) CreateAttach (p. 1053)

TS_ConstId ::= SEQUENCE {

ts_ConstIdentifier	Identifier	--See page 1100

}

Referenced from: TS_ConstDcl (p. 1035) TS_ConstRef (p. 1035)

TS_ConstType ::= SEQUENCE {

ttcn_Type	TTCN_Type	--See page 1089

}

Referenced from: TS_ConstDcl (p. 1035) TS_ConstRef (p. 1035)

TS_ConstValue ::= SEQUENCE {

declarationValue	DeclarationValue	--See page 1054

}

Referenced from: TS_ConstDcl (p. 1035) TS_ConstRef (p. 1035)

TS_OpId ::= SEQUENCE {
    ts_OpIdAndParList TS_OpIdAndParList --See page 1088
}
Referenced from: TS_OpDef (p. 1036)

TS_OpIdAndParList ::= SEQUENCE {
    ts_OpIdentifier Identifier --See page 1100
    formalParList FormalParList --See page 1060
}
Referenced from: TS_OpId (p. 1088)

TS_OpResult ::= SEQUENCE {
    ttcn_Type TTCN_Type --See page 1089
}
Referenced from: TS_OpDef (p. 1036)

TS_ParId ::= SEQUENCE {
    ts_ParIdentifier Identifier --See page 1100
}
Referenced from: TS_ParDcl (p. 1036)

TS_ParType ::= SEQUENCE {
    ttcn_Type TTCN_Type --See page 1089
}
Referenced from: TS_ParDcl (p. 1036)

TS_ProcId ::= SEQUENCE {
    ts_ProcIdAndParList TS_ProcIdAndParList --See page 1088
}
Referenced from: TS_ProcDef (p. 1037)

TS_ProcIdAndParList ::= SEQUENCE {
    ts_ProcIdentifier Identifier --See page 1100
    formalParList FormalParList --See page 1060
}
Referenced from: TS_ProcId (p. 1088)

TS_ProcResult ::= SEQUENCE {

ttcn_Type	TTCN_Type	--See page 1089

}

Referenced from: TS_ProcDef (p. 1037)

TS_VarId ::= SEQUENCE {

ts_VarIdentifier	Identifier	--See page 1100

}

Referenced from: TS_VarDcl (p. 1038)

TS_VarType ::= SEQUENCE {

ttcn_Type	TTCN_Type	--See page 1089

}

Referenced from: TS_VarDcl (p. 1038)

TS_VarValue ::= SEQUENCE {

declarationValue	DeclarationValue	--See page 1054

}

Referenced from: TS_VarDcl (p. 1038)

TTCN_Type ::= CHOICE {

predefinedType	PredefinedType	--See page 1073
referenceType	ReferenceType	--See page 1075

}

Referenced from: TypeAndRestriction (p. 1090) TypeAndLengthAttribute (p. 1090) TS_OpResult (p. 1088) TS_ProcResult (p. 1089) TS_ParType (p. 1088) TS_ConstType (p. 1087) TS_VarType (p. 1089) TC_VarType (p. 1084) FormalParType (p. 1061)

TypeAndAttributes ::= CHOICE {

typeAndLengthAttribute	TypeAndLengthAttribute	--See page 1090
pdu	PDU	--See page 1104

}

Referenced from: ElemType (p. 1057) ASP_ParType (p. 1045) PDU_FieldType (p. 1072) CM_ParType (p. 1049)

TypeAndLengthAttribute ::= SEQUENCE {

ttcn_Type	TTCN_Type		--See page 1089
lengthAttribute	LengthAttribute	OPTIONAL	--See page 1063

}

Referenced from: TypeAndAttributes (p. 1089)

TypeAndRestriction ::= SEQUENCE {

ttcn_Type	TTCN_Type		--See page 1089
restriction	Restriction	OPTIONAL	--See page 1077

}

Referenced from: SimpleTypeDefinition (p. 1080)

TypeAssignment ::= SEQUENCE {

typeReference	TypeReference	--See page 1090
asn1_Type	ASN1_Type	--See page 1043

}

Referenced from: TypeAssignmentList (p. 1090)

TypeAssignmentList ::= SEQUENCE OF {

typeAssignment	TypeAssignment	--See page 1090

}

Referenced from: ASN1_LocalTypes (p. 1043)

TypeConstraints ::= SEQUENCE {

namedConstraintList	NamedConstraintList	--See page 1066

}

Referenced from: FullSpecification (p. 1061) PartialSpecification (p. 1070)

TypeReference ::= SEQUENCE {

identifier	Identifier	--See page 1100

}

Referenced from: TypeAssignment (p. 1090)

UnaryExpression ::= SEQUENCE {

unaryOp	UnaryOp	--See page 1091
primary	Primary	--See page 1074

}

Referenced from: Factor (p. 1060)

UnaryOp ::= CHOICE {

plus	Plus		--See page 1105
minus	Minus		--See page 1102
not	Not		--See page 1103

}

Referenced from: UnaryExpression (p. 1090)

Unit ::= SEQUENCE {

timeUnit	TimeUnit		--See page 1087

}

Referenced from: TimerDcl (p. 1034)

UpperBound ::= CHOICE {

bound	Bound		--See page 1047
infinity	INFINITY		--See page 1101

}

Referenced from: RangeLength (p. 1074)

UpperEndpoint ::= SEQUENCE {

upperEndValue	UpperEndValue		--See page 1091
inclusive	INCLUSIVE	OPTIONAL	--See page 1101

}

Referenced from: ValueRange (p. 1093)

UpperEndValue ::= CHOICE {

constraintValueAndAttributes	ConstraintValueAndAttributes	--See page 1052
max	Max	--See page 1102

}

Referenced from: UpperEndpoint (p. 1091)

UpperRangeBound ::= CHOICE {

constraintExpression	ConstraintExpression	--See page 1051
infinity	INFINITY	--See page 1101

}

Referenced from: ValRange (p. 1092)

UpperTypeBound ::= CHOICE {

signedNumber	SignedNumber	--See page 1079
number	Number	--See page 1103
infinity	INFINITY	--See page 1101

}

Referenced from: RangeTypeLength (p. 1074) IntegerRange (p. 1062)

UpperValueBound ::= CHOICE {

valueBound	ValueBound	--See page 1093
infinity	INFINITY	--See page 1101

}

Referenced from: RangeValueLength (p. 1074)

UsefulType ::= CHOICE {

generalizedTime	GeneralizedTime	--See page 1099
utcTime	UTCTime	--See page 1108
external	EXTERNAL	--See page 1098
objectDescriptor	ObjectDescriptor	--See page 1103

}

Referenced from: BuiltinType (p. 1047)

ValRange ::= SEQUENCE {

lowerRangeBound	LowerRangeBound	--See page 1064
upperRangeBound	UpperRangeBound	--See page 1091

}

Referenced from: MatchingSymbol (p. 1065)

Value ::= CHOICE {

literalValue	LiteralValue	--See page 1064
asn1_Value	ASN1_Value	--See page 1044

}

Referenced from: Primary (p. 1074)

ValueAttributes ::= SEQUENCE {

valueLength	ValueLength	OPTIONAL	--See page 1093
if_PRESENT	IF_PRESENT	OPTIONAL	--See page 1101

}

Referenced from: ConstraintValueAndAttributes (p. 1052)

ValueBound ::= CHOICE {

number	Number	--See page 1103
identifier	Identifier	--See page 1100

}

Referenced from: SingleValueLength (p. 1081) LowerValueBound (p. 1065) UpperValueBound (p. 1092)

ValueConstraint ::= SEQUENCE {

subTypeSpec	SubTypeSpec	--See page 1082

}

Referenced from: Constraint (p. 1051)

ValueLength ::= CHOICE {

singleValueLength	SingleValueLength	--See page 1081
rangeValueLength	RangeValueLength	--See page 1074

}

Referenced from: ValueAttributes (p. 1092)

ValueList ::= SEQUENCE OF {

constraintValueAndAttributes	ConstraintValueAndAttributes	--See page 1052

}

Referenced from: MatchingSymbol (p. 1065) Complement (p. 1049) Permutation (p. 1073)

ValueRange ::= SEQUENCE {

lowerEndpoint	LowerEndpoint	--See page 1064
upperEndpoint	UpperEndpoint	--See page 1091

}

Referenced from: SubtypeValueSet (p. 1082)

VariationDefault ::= SEQUENCE {

expression	Expression	--See page 1060

}

Referenced from: EncodingVariation (p. 1023)

VariationRef ::= SEQUENCE {

variationReference	VariationReference	--See page 1108

}

Referenced from: EncodingVariation (p. 1023)

Verdict ::= CHOICE {

pass	Pass	--See page 1070
fail	Fail	--See page 1060
inconclusive	Inconclusive	--See page 1062
result	Result	--See page 1077

}

Referenced from: VerdictId (p. 1094)

VerdictId ::= SEQUENCE {

verdict	Verdict	--See page 1094

}

Referenced from: BehaviourLine (p. 1017)

WithComponent ::= SEQUENCE {

singleTypeConstraint	SingleTypeConstraint	--See page 1080

}

Referenced from: InnerTypeConstraints (p. 1062)

WithComponents ::= SEQUENCE {

multipleTypeConstraints	MultipleTypeConstraints	--See page 1065

}

Referenced from: InnerTypeConstraints (p. 1062)

Terminal Nodes

This section contains all the Terminal Nodes. A Terminal Node is defined to be a node without children in the Parse Tree, and they generally represents some actual piece of text in the field. The Identifier class is a good example of a Terminal Node. It is the Access representation of a name or identifier. The Terminal Nodes have some special properties, they can be printed directly, yielding the piece of text they represent, and their positions in the field string can be easily found. The Identifier is a special Terminal Node since all Identifiers have an type which is information about in what table the identifier was declared (or Unknown if this could not be found during analysis).

ABSENT ::= TERMINAL
Referenced from: PresenceConstraint (p. 1073)

AEXPLICIT ::= TERMINAL
Referenced from: TagAttributes (p. 1083)

AFalse ::= TERMINAL
Referenced from: BooleanValue (p. 1046)

AIMPLICIT ::= TERMINAL
Referenced from: TagAttributes (p. 1083)

And ::= TERMINAL
Referenced from: MultOp (p. 1066)

ANY ::= TERMINAL
Referenced from: AnyType (p. 1042)

AnyOrOmit ::= TERMINAL
Referenced from: MatchingSymbol (p. 1065)

ANYValue ::= TERMINAL
Referenced from: MatchingSymbol (p. 1065)

APPLICATION ::= TERMINAL
Referenced from: ClassType (p. 1048)

ASN1_ModuleId ::= TERMINAL
Referenced from: ASN1_TypeRef (p. 1015) TS_ConstRef (p. 1035) ASN1_ASP_TypeDefByRef (p. 1011) ASN1_PDU_TypeDefByRef (p. 1013)

ASN1_TypeReference ::= TERMINAL
Referenced from: ASN1_TypeRef (p. 1015) ASN1_ASP_TypeDefByRef (p. 1011) ASN1_PDU_TypeDefByRef (p. 1013)

ASN1_ValueReference ::= TERMINAL
Referenced from: TS_ConstRef (p. 1035)

ATrue ::= TERMINAL
Referenced from: BooleanValue (p. 1046)

BITSTRING ::= TERMINAL
Referenced from: PredefinedType (p. 1073)

BOOLEAN ::= TERMINAL
Referenced from: PredefinedType (p. 1073)

BooleanType ::= TERMINAL
Referenced from: BuiltinType (p. 1047)

Bstring ::= TERMINAL
Referenced from: LiteralValue (p. 1064)

Comment ::= TERMINAL

Referenced from: ASN1_CM_Constraint (p. 1012) CompactTestCase (p. 1018) TestCase (p. 1032) BehaviourLine (p. 1017) TestStep (p. 1033) DefaultCase (p. 1020) SuiteStructure (p. 1030) EncodingDefinition (p. 1022) EncodingVariationSet (p. 1024) EncodingVariation (p. 1023) InvalidFieldEncodingDef (p. 1024) SimpleTypeDef (p. 1027) StructTypeDef (p. 1030) ElemDcl (p. 1022) ASN1_TypeDef (p. 1014) ASN1_TypeRef (p. 1015) TS_OpDef (p. 1036) TS_ProcDef (p. 1037) TS_ParDcl (p. 1036) SelectExprDef (p. 1027) TS_ConstDcl (p. 1035) TS_ConstRef (p. 1035) TS_VarDcl (p. 1038) TC_VarDcl (p. 1030) TCompDcl (p. 1032) TCompConfigDcl (p. 1031) TCompConfigInfo (p. 1031) PCO_TypeDcl (p. 1025) PCO_Dcl (p. 1025) CP_Dcl (p. 1019) TimerDcl (p. 1034) TTCN_ASP_TypeDef (p. 1039) ASP_ParDcl (p. 1015) ASN1_ASP_TypeDef (p. 1011) ASN1_ASP_TypeDefByRef (p. 1011) TTCN_PDU_TypeDef (p. 1040) PDU_FieldDcl (p. 1026) ASN1_PDU_TypeDef (p. 1013) ASN1_PDU_TypeDefByRef (p. 1013) TTCN_CM_TypeDef (p. 1039) CM_ParDcl (p. 1017) ASN1_CM_TypeDef (p. 1012) AliasDef (p. 1010) StructTypeConstraint (p. 1029) ElemValue (p. 1022) ASN1_TypeConstraint (p. 1014) TTCN_ASP_Constraint (p. 1038) ASP_ParValue (p. 1016) TTCN_PDU_Constraint (p. 1040) PDU_FieldValue (p. 1026) ASN1_ASP_Constraint (p. 1010) ASN1_PDU_Constraint (p. 1012) TTCN_CM_Constraint (p. 1039) CM_ParValue (p. 1018)

CP ::= TERMINAL

Referenced from: FormalParType (p. 1061)

Cstring ::= TERMINAL

Referenced from: LiteralValue (p. 1064)

DefaultGroupReference ::= TERMINAL

Referenced from: DefaultRef (p. 1055)

Description ::= TERMINAL

Referenced from: CompactTestCase (p. 1018) TestCase (p. 1032) TestStep (p. 1033) DefaultCase (p. 1020)

DetailedComment ::= TERMINAL
Referenced from: SuiteStructure (p. 1030) TestCaseIndex (p. 1032) TestStepIndex (p. 1034) DefaultIndex (p. 1021) EncodingDefinitions (p. 1023) EncodingVariationSet (p. 1024) InvalidFieldEncodingDef (p. 1024) SimpleTypeDefs (p. 1028) StructTypeDef (p. 1030) ASN1_TypeDef (p. 1014) ASN1_TypeRefs (p. 1015) TS_OpDef (p. 1036) TS_ProcDef (p. 1037) TS_ParDcls (p. 1037) SelectExprDefs (p. 1027) TS_ConstDcls (p. 1035) TS_ConstRefs (p. 1036) TS_VarDcls (p. 1038) TC_VarDcls (p. 1031) TCompDcls (p. 1032) TCompConfigDcl (p. 1031) PCO_TypeDcls (p. 1025) PCO_Dcls (p. 1025) CP_Dcls (p. 1020) TimerDcls (p. 1035) TTCN_ASP_TypeDef (p. 1039) ASN1_ASP_TypeDef (p. 1011) ASN1_ASP_TypeDefsByRef (p. 1011) TTCN_PDU_TypeDef (p. 1040) ASN1_PDU_TypeDef (p. 1013) ASN1_PDU_TypeDefsByRef (p. 1014) TTCN_CM_TypeDef (p. 1039) ASN1_CM_TypeDef (p. 1012) AliasDefs (p. 1010) StructTypeConstraint (p. 1029) ASN1_TypeConstraint (p. 1014) TTCN_ASP_Constraint (p. 1038) TTCN_PDU_Constraint (p. 1040) ASN1_ASP_Constraint (p. 1010) ASN1_PDU_Constraint (p. 1012) TTCN_CM_Constraint (p. 1039) ASN1_CM_Constraint (p. 1012) TestCase (p. 1032) TestStep (p. 1033) DefaultCase (p. 1020)

Div ::= TERMINAL
Referenced from: MultOp (p. 1066)

Encoding_TypeList ::= TERMINAL
Referenced from: EncodingVariationSet (p. 1024) InvalidFieldEncodingDef (p. 1024)

EncodingReference ::= TERMINAL
Referenced from: EncodingRef (p. 1057)

Equal ::= TERMINAL
Referenced from: RelOp (p. 1076)

EXTERNAL ::= TERMINAL
Referenced from: UsefulType (p. 1092)

FAIL ::= TERMINAL
Referenced from: Fail (p. 1060) R_Value (p. 1074)

FullIdentifier ::= TERMINAL
Referenced from: StructIdAndFullId (p. 1082) ElemIdAndFullId (p. 1057) ASN1_TypeId (p. 1044) ASP_IdAndFullId (p. 1044) ASP_ParIdAndFullId (p. 1045) PDU_IdAndFullId (p. 1073) PDU_FieldIdAndFullId (p. 1072) CM_ParIdAndFullId (p. 1049)

GeneralizedTime ::= TERMINAL
Referenced from: UsefulType (p. 1092)

GeneralString ::= TERMINAL
Referenced from: CharacterString (p. 1048)

GraphicString ::= TERMINAL
Referenced from: CharacterString (p. 1048)

Greater ::= TERMINAL
Referenced from: RelOp (p. 1076)

GreaterOrEqual ::= TERMINAL
Referenced from: RelOp (p. 1076)

HEXSTRING ::= TERMINAL
Referenced from: PredefinedType (p. 1073)

Hstring ::= TERMINAL
Referenced from: LiteralValue (p. 1064)

IA5String ::= TERMINAL
Referenced from: CharacterString (p. 1048)

Identifier ::= TERMINAL

Referenced from: Send (p. 1078) ImplicitSend (p. 1061) Receive (p. 1075) Receive (p. 1075) Otherwise (p. 1069) Timeout (p. 1086) TCompIdList (p. 1084) StartTimer (p. 1081) CancelTimer (p. 1047) ReadTimer (p. 1075) TreeReference (p. 1087) CreateTComp (p. 1053) ConstraintReference (p. 1051) ConsRef (p. 1051) TreeHeader (p. 1087) TestGroupId (p. 1085) TestStepGroupId (p. 1085) TestStepIdAndParList (p. 1086) DefaultGroupId (p. 1054) DefaultIdAndParList (p. 1054) ValueBound (p. 1093) DataObjectReference (p. 1054) ComponentIdentifier (p. 1049) BitIdentifier (p. 1046) OpCall (p. 1069) TypeReference (p. 1090) DefinedType (p. 1055) NamedNumber (p. 1066) IdOrNum (p. 1061) NamedType (p. 1067) SelectionType (p. 1077) SuiteId (p. 1083) SO_SuiteId (p. 1081) EncVariationIdAndParList (p. 1058) InvalidFieldEncodingIdAndParList (p. 1063) SimpleTypeId (p. 1080) ReferenceType (p. 1075) AnyDefinedBy (p. 1042) EncVariationCall (p. 1058) NamedConstraint (p. 1066) InvalidFieldEncodingCall (p. 1062) IdentifierList (p. 1061) EncodingRuleId (p. 1057) NamedValue (p. 1067) NameForm (p. 1067) DefinedValue (p. 1055) FormalParIdList (p. 1060) EncRuleId (p. 1058) PDU_EncodingId (p. 1071) StructIdAndFullId (p. 1082) ElemIdAndFullId (p. 1057) Bound (p. 1047) ASN1_TypeId (p. 1044) TS_OpIdAndParList (p. 1088) TS_ProcIdAndParList (p. 1088) TS_ParId (p. 1088) SelectExprId (p. 1077) TS_ConstId (p. 1087) TS_VarId (p. 1089) TC_VarId (p. 1083) TCompId (p. 1084) TCompConfigId (p. 1084) TCompUsed (p. 1084) PCO_List (p. 1070) CP_List (p. 1053) PCO_Id (p. 1070) PCO_TypeId (p. 1071) CP_Id (p. 1053) TimerId (p. 1086) ASP_IdAndFullId (p. 1044) PCO_Type (p. 1071) ASP_ParIdAndFullId (p. 1045) PDU_IdAndFullId (p. 1073) PDU_FieldIdAndFullId (p. 1072) CM_Id (p. 1049) CM_ParIdAndFullId (p. 1049) AliasId (p. 1042) Expansion (p. 1059) ConsIdAndParList (p. 1051) DerivationPath (p. 1056) TestCaseId (p. 1085) Configuration (p. 1050) SelExprId (p. 1078) DefaultReference (p. 1055) Label (p. 1063) Send (p. 1078)

Impl. Note: Every Identifier has an associated type, which refers to the context where the Identifier was declared. If, for example, an Identifier refers to a TTCN ASP declaration table, then its type will be Choices::_TTCN_ASP, and if the Identifier refers to a parameter of a constraint or test step, then its type will be Choices::_FormalParam_Id. The associated type is reached through the member function Identifier::choice().

Identifier types: Alias ASN1_ASP ASN1_ASP_Cons ASN1_Cons ASN1_CM ASN1_CM_Cons ASN1_PDU ASN1_PDU_Cons ASN1_LocalType ASN1_NamedType ASN1_NameForm ASN1_Type ASP_Par BitString_NamedNumber CM_Par CP DefaultCase DefaultGroup Elem Enumeration_NamedNumber FormalParam Integer_NamedNumber Label LocalTree PCO PCO_Type PredefinedOp PDU_Field Result SelectExpr SimpleType StructType StructCons Suite TC_Var TComp TCompConfig TestCase TestStep TestStepGroup TestGroup Timer TS_Const TS_Op TS_Par TS_Var TTCN_ASP TTCN_ASP_Cons TTCN_CM TTCN_CM_Cons TTCN_PDU TTCN_PDU_Cons Unknown Verdict

IF_PRESENT ::= TERMINAL
Referenced from: ValueAttributes (p. 1092)

INCLUSIVE ::= TERMINAL
Referenced from: LowerEndpoint (p. 1064) UpperEndpoint (p. 1091)

INCONC ::= TERMINAL
Referenced from: Inconclusive (p. 1062) R_Value (p. 1074)

INFINITY ::= TERMINAL
Referenced from: UpperTypeBound (p. 1092) UpperBound (p. 1091) UpperRangeBound (p. 1091) UpperValueBound (p. 1092)

INTEGER ::= TERMINAL
Referenced from: PredefinedType (p. 1073)

InvalidFieldEncodingDefinition ::= TERMINAL
Referenced from: InvalidFieldEncodingDef (p. 1024)

ISO646String ::= TERMINAL
Referenced from: CharacterString (p. 1048)

Less ::= TERMINAL
Referenced from: RelOp (p. 1076)

LessOrEqual ::= TERMINAL
Referenced from: RelOp (p. 1076)

LT ::= TERMINAL
Referenced from: PCO_Role (p. 1071)

MacroSymbol ::= TERMINAL
Referenced from: ASP_ParIdOrMacro (p. 1045) PDU_FieldIdOrMacro (p. 1072) CM_ParIdOrMacro (p. 1049)

Max ::= TERMINAL
Referenced from: UpperEndValue (p. 1091)

MicroSeconds ::= TERMINAL
Referenced from: TimeUnit (p. 1087)

MilliSeconds ::= TERMINAL
Referenced from: TimeUnit (p. 1087)

Min ::= TERMINAL
Referenced from: LowerEndValue (p. 1064)

Minus ::= TERMINAL
Referenced from: AddOp (p. 1041) UnaryOp (p. 1091) SignedNumber (p. 1079)

Minus_INFINITY ::= TERMINAL
Referenced from: SpecialRealValue (p. 1081)

MinusINFINITY ::= TERMINAL
Referenced from: LowerTypeBound (p. 1064) LowerRangeBound (p. 1064)

Minutes ::= TERMINAL
Referenced from: TimeUnit (p. 1087)

Mod ::= TERMINAL
Referenced from: MultOp (p. 1066)

MTC ::= TERMINAL
Referenced from: TComp_Role (p. 1084)

Mult ::= TERMINAL
Referenced from: MultOp (p. 1066)

NanoSeconds ::= TERMINAL
Referenced from: TimeUnit (p. 1087)

NONE ::= TERMINAL
Referenced from: R_Value (p. 1074)

Not ::= TERMINAL
Referenced from: UnaryOp (p. 1091)

NotEqual ::= TERMINAL
Referenced from: RelOp (p. 1076)

NullType ::= TERMINAL
Referenced from: BuiltinType (p. 1047)

NullValue ::= TERMINAL
Referenced from: ASN1_Value (p. 1044)

Number ::= TERMINAL
Referenced from: SingleTypeLength (p. 1080) LowerTypeBound (p. 1064) UpperTypeBound (p. 1092) Bound (p. 1047) Num_PCOs (p. 1068) Num_CPs (p. 1068) LineNumber (p. 1063) Indentation (p. 1062) ValueBound (p. 1093) ComponentPosition (p. 1050) LiteralValue (p. 1064) IdOrNum (p. 1061) ClassNumber (p. 1048) NumberForm (p. 1068) Mantissa (p. 1065) Exponent (p. 1059) SignedNumber (p. 1079)

NumericString ::= TERMINAL
Referenced from: CharacterString (p. 1048)

ObjectDescriptor ::= TERMINAL
Referenced from: UsefulType (p. 1092)

ObjectIdentifierType ::= TERMINAL
Referenced from: BuiltinType (p. 1047)

Objective ::= TERMINAL
Referenced from: TestGroup (p. 1033) CompactTestGroup (p. 1019) TestStep (p. 1033) DefaultCase (p. 1020)

OCTETSTRING ::= TERMINAL
Referenced from: PredefinedType (p. 1073)

OctetStringType ::= TERMINAL
Referenced from: BuiltinType (p. 1047)

OMIT ::= TERMINAL
Referenced from: Omit (p. 1069)

OPTIONAL ::= TERMINAL
Referenced from: NamedTypeAttribute (p. 1067) PresenceConstraint (p. 1073)

Or ::= TERMINAL
Referenced from: AddOp (p. 1041)

Ostring ::= TERMINAL
Referenced from: LiteralValue (p. 1064)

PASS ::= TERMINAL
Referenced from: Pass (p. 1070) R_Value (p. 1074)

PDU ::= TERMINAL
Referenced from: TypeAndAttributes (p. 1089) FormalParType (p. 1061)

PicoSeconds ::= TERMINAL
Referenced from: TimeUnit (p. 1087)

PICS_PIXITref ::= TERMINAL
Referenced from: TS_ParDcl (p. 1036)

Plus ::= TERMINAL
Referenced from: AddOp (p. 1041) UnaryOp (p. 1091)

Plus_INFINITY ::= TERMINAL
Referenced from: SpecialRealValue (p. 1081)

PreliminaryFAIL ::= TERMINAL
Referenced from: Fail (p. 1060)

PreliminaryINCONC ::= TERMINAL
Referenced from: Inconclusive (p. 1062)

PreliminaryPASS ::= TERMINAL
Referenced from: Pass (p. 1070)

PRESENT ::= TERMINAL
Referenced from: PresenceConstraint (p. 1073)

PrintableString ::= TERMINAL
Referenced from: CharacterString (p. 1048)

PRIVATE_ ::= TERMINAL
Referenced from: ClassType (p. 1048)

PTC ::= TERMINAL
Referenced from: TComp_Role (p. 1084)

R ::= TERMINAL
Referenced from: Result (p. 1077)

R_TYPE ::= TERMINAL
Referenced from: PredefinedType (p. 1073)

RealType ::= TERMINAL
Referenced from: BuiltinType (p. 1047)

RETURN_ ::= TERMINAL
Referenced from: Construct (p. 1052)

Seconds ::= TERMINAL
Referenced from: TimeUnit (p. 1087)

SO_DefaultId ::= TERMINAL
Referenced from: SO_DefIndex (p. 1028)

SO_DefaultRef ::= TERMINAL
Referenced from: SO_DefIndex (p. 1028)

SO_Description ::= TERMINAL
Referenced from: SO_CaseIndex (p. 1028) SO_StepIndex (p. 1028) SO_DefIndex (p. 1028)

SO_Objective ::= TERMINAL
Referenced from: SO_StructureAndObjective (p. 1029)

SO_PICSRef ::= TERMINAL
Referenced from: SuiteStructure (p. 1030)

SO_PIXITRef ::= TERMINAL
Referenced from: SuiteStructure (p. 1030)

SO_SelExprId ::= TERMINAL
Referenced from: SO_StructureAndObjective (p. 1029) SO_CaseIndex (p. 1028)

SO_StandardsRef ::= TERMINAL
Referenced from: SuiteStructure (p. 1030)

SO_TestCaseId ::= TERMINAL
Referenced from: SO_CaseIndex (p. 1028)

SO_TestGroupRef ::= TERMINAL
Referenced from: SO_StructureAndObjective (p. 1029) SO_CaseIndex (p. 1028)

SO_TestMethods ::= TERMINAL
Referenced from: SuiteStructure (p. 1030)

SO_TestStepId ::= TERMINAL
Referenced from: SO_StepIndex (p. 1028)

SO_TestStepRef ::= TERMINAL
Referenced from: SO_StepIndex (p. 1028)

T61String ::= TERMINAL
Referenced from: CharacterString (p. 1048)

TeletexString ::= TERMINAL
Referenced from: CharacterString (p. 1048)

Ten ::= TERMINAL
Referenced from: Base (p. 1046)

TestGroupReference ::= TERMINAL
Referenced from: TestGroupRef (p. 1085)

TestPurpose ::= TERMINAL
Referenced from: CompactTestCase (p. 1018) TestCase (p. 1032)

TestStepGroupReference ::= TERMINAL
Referenced from: TestStepRef (p. 1086)

TIMER ::= TERMINAL
Referenced from: FormalParType (p. 1061)

TS_OpDescription ::= TERMINAL
Referenced from: TS_OpDef (p. 1036)

TS_ProcDescription ::= TERMINAL
Referenced from: TS_ProcDef (p. 1037)

Two ::= TERMINAL
Referenced from: Base (p. 1046)

UNIVERSAL ::= TERMINAL
Referenced from: ClassType (p. 1048)

UT ::= TERMINAL
Referenced from: PCO_Role (p. 1071)

UTCTime ::= TERMINAL
Referenced from: UsefulType (p. 1092)

VariationReference ::= TERMINAL
Referenced from: VariationRef (p. 1093)

VideotexString ::= TERMINAL
Referenced from: CharacterString (p. 1048)

VisibleString ::= TERMINAL
Referenced from: CharacterString (p. 1048)

Chapter

24

The TTCN Browser (on UNIX)

This chapter contains a reference manual to the TTCN Browser. The Browser presents an overview of the TTCN document. In the Browser, it is possible to edit the TTCN document structure, to determine which parts of the document to view, and to apply different tools.

The functionality of the Browser, the menus, windows and quick buttons, are described in this chapter. The TTCN Table Editor is described in chapter 25, *The TTCN Table Editor (on UNIX)*. How to analyze and find tables is described in chapter 26, *Analyzing TTCN Documents (on UNIX)*.

For an overview of the TTCN Suite, see chapter 3, *Introduction to the TTCN Suite (on UNIX), in the TTCN Suite 6.2 Getting Started*.

> **Note: UNIX version**
>
> **This is the UNIX version of the chapter. The corresponding Windows chapter is chapter 30, *Editing TTCN Documents (in Windows)*.**

Introduction to the TTCN Browser

The TTCN Browser is the part of the TTCN Suite that presents TTCN document structures. When a new TTCN document is opened, place holders for the static parts of the document are already created. Then you can add and delete the dynamic parts.

The amount of information actually displayed in a Browser can be collapsed and expanded. It is also possible to display selected parts of the TTCN document in a sub Browser. You open a sub Browser from the main Browser or from another sub Browser.

The functionality of a menu choice is applied to all items or selections of items in the Browser. If an operation is invoked from the Organizer (see chapter 2, *The Organizer*), it is always applied to all items in the TTCN document.

Opening the Browser and a TTCN Document

You open the TTCN Browser by selecting a TTCN document already included in the Organizer followed by the menu choice *Edit*, or by double-clicking the TTCN document. You may also add a new TTCN document with *Add New* or *Add Existing* from the *Edit* menu, or start a Browser directly by selecting *Editors > TTCN Browser* from the *Tools* menu. See chapter 2, *The Organizer* for more information.

If you try to open a TTCN document that is already opened by any another user or in another TTCN Suite Browser, you will get a question informing that the file is opened by another user and ask to open it in read-only mode. In read-only mode you will not be able to save your changes (if any) in the same file, but free to save the TTCN document as another file. No other restrictions exist in read-only mode: you may browse, analyze, generate code, etc.

The question that the file is opened by another user is isuued because a read-lock file (with additional suffix '-read_lock') exists in the same directory as the document. This may also happen if the TTCN Suite or Windows has crashed. In this case just remove this file and try to open the TTCN document again.

Close the Browser and a TTCN Document

To close the Browser and all corresponded table editors, in the Browser *File* menu choose *Close*. The TTCN Suite will remain running, TTCN documents remains loaded and the license is not released in this case. The *Close* command means only that the Browser window is closed. To exit TTCN Suite Editor, go to the Browser *File* menu and choose *Exit*.

Exit the TTCN Suite Editor

To close all Browsers and table editors, you can in any Browser's *File* menu choose *Exit*. The TTCN Suite will then close all Browsers, release the license and finish. If any Browser contains any modified TTCN documents, you will be prompted to save your document.

The Browser User Interface

For a general description of the user interface in SDL Suite and TTCN Suite, see chapter 1, *User Interface and Basic Operations*.

The TTCN document in the Browser contains *items*. These are:

- *Static items*, common to every TTCN document, such as the declarations part or constraints part.
- *Dynamic items* – that is TTCN objects – such as tables, groups or objects in a table, for example test case variables.

The Browser displays all the static items in the order defined in the TTCN standard. This ordering cannot be changed. For example, it is not possible to have the constraints part coming before the declarations part. Only the dynamic items – the tables – may be added and deleted. The structure is indicated by indention.

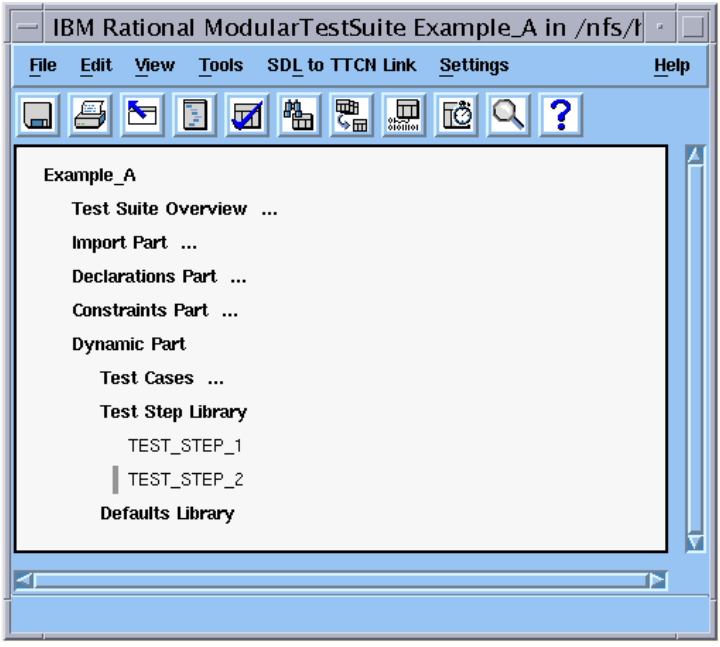


Figure 183: The the TTCN Browser window **on UNIX**

Figure 183 depicts the Browser window. The test suite overview, the import part, the declarations part, the constraints part, the test cases and the defaults library are collapsed. This is indicated by the three dots.

The dynamic part and the test step library are expanded. The test step library contains two test steps: `LT` and `UT`. `LT` has been edited but not analyzed. This is indicated by the shaded bar (gray on a color screen). The table icon indicates that `UT` is opened and the filled bar (red on a color screen) that `UT` has been analyzed and is erroneous. Items with no analysis bars have been analyzed and contain no errors.

Items in **bold** font represent the static (or structural) parts of the TTCN document and cannot be opened or edited. The items in normal font represent named TTCN tables which can be opened and edited.

Some of the static items may remain unused. For example, if *Test Case Variables* is not used it will still be displayed in the Browser. However, it will not be printed or output in the TTCN-MP format.

Controlling the Items in the Browser

Selecting Items

There is a difference between *selection* and *input focus*. Selection is an inverted item or text string. The input focus is indicated by a rectangular border surrounding the relevant item. Keyboard commands operate on the item with input focus, not the selection.

Point and click once to select a **single** item in a Browser – note that this also sets the input focus. Selecting a new item will deselect the previous item (if any).

To select **several** items, press `<Ctrl>` while selecting items. Note that the input focus is set to the last selected item. Pressing `<Ctrl>` while selecting items works in toggle mode: a selected item will be deselected and vice versa.

To select a range of items, select the first, and then press `<Shift>` and click on the last item in the range.

When clicking the **middle** mouse button, the input focus will be set to the new item. The selection will not be changed.

You can also select items by using:

- The popup menus, see "Using Popup Menus" on page 1123.
- Selector, see "Using More Complex Selections" on page 1124.
- Compare, see "Comparing TTCN Documents" on page 1137.

> **Hint:**
>
> Some of the commands in the TTCN Suite only work when a single item is selected in a Browser. Before applying such commands, it is recommended that you deselect all items before you select a single item.

Controlling What is Displayed

The *View* menu is used for deciding how much of the test suite that should be displayed in a Browser. This is useful when you work with large test suites. This menu also contains commands to access any opened Table Editor and to close all opened editors, as well as to access any loaded TTCN document.

View > Collapse

Hides all items under the marked item. A collapsed item is indicated by an ellipsis (...). A shortcut for *Collapse* is to type `<c>` without invoking the *View* menu.

View > Expand

Displays all items at the next level below the marked item. A short-cut for *Expand* is to type `<e>` without invoking the *View* menu.

View > Expand Tree

Displays all items at all levels below the marked item. A short-cut for *Expand Tree* is to type `<Shift+e>` without invoking the *View* menu.

View > Close All Editors

Closes all open Table Editors, regardless of which document they belong to.

View > Show Editor

This sub menu presents a list of all open Table Editors, in any TTCN document. By selecting an item in this list, the corresponding Table Editor is brought forward.

View > Show Document

This sub menu presents a list of all loaded TTCN documents. By selecting an item in this list, the main Browser of the corresponding document is brought forward.

Creating a Sub Browser

Sub Browsers may be created to display selected portions of a test suite. These selections may be made manually or with the Selector (see above).

This facility is useful when working with a large test suite or with items that fulfil certain criteria.

Tools > Browser

Causes a sub Browser to be displayed containing only the selected items from this Browser. There is no limit on the number of sub Browsers that may be displayed.

> **Note:**
> Closing the main Browser also closes any sub Browsers that are opened.

Editing the TTCN Document Structure

The TTCN Suite displays all the components of a TTCN document in the structured format of the Browser. Part of this structure is static and common to every TTCN document. This static structure is automatically generated when the TTCN document is first created.

To build an individual TTCN document, named tables (such as PDUs, Constraints and Behaviours) and named objects to the multi-object tables (such as Test Case Variables, PCOs and Timers) must be added to this static structure.

The *Edit* menu is used to add, delete, copy, cut, paste etc. *editable items* (i.e. groups, tables or objects in a multiple object table) in a Browser.

Adding and Inserting Items

Edit > Add

Adds a new table to the Browser or adds an object to a table in the Browser.

In the case of single-object tables, select the Browser item that "holds" the table. Choosing *Add* will add a new table to the end of the list of tables. The new table will be unnamed. Use the *Rename* command to give the table its correct name, or open the table and edit the name there.

Example 189 ––

To add a new TTCN PDU to a Browser, select the item *TTCN PDU Type Definitions* and apply *Add*.

––

In the case of multiple-object tables, select the item that is the name of the table. Choosing *Add* will add a new object to the end of the list of objects in this table. The new object will be unnamed. Use the *Rename* command to give the table its correct name, or open the table and edit the name there.

Example 190 ––

To add a new Test Case Variable to a Browser, select the item *Test Case Variables* item and apply *Add*.

––

Edit > Insert Before

Similar to *Add* but the new table (or object in a table) is inserted **before** a selected table (or selected object).

Note:

Only a single item should be selected.

Edit > Insert After

Similar to *Add* but the new table (or object in a table) is inserted **after** a selected table (or selected object).

Note:

Only a single item should be selected.

Adding and Inserting Groups

The following commands are used to add/insert new groups to the Test Case, Test Step and Default libraries.

Edit > Add Group

Adds a new Test (Case) Group, a new Test Step Group, a new Default Group in the Dynamic Part, or a new Group in Constraints Part. The new group is added at the **next level after** the selected group.

Use the *Rename* command to edit the temporary name *NoName*. Note that the trailing slash (/) is automatically added.

To add a table to a group, select the group and use the *Add* command.

Edit > Insert Group Before

Inserts a new group **before** a selected item.

Edit > Insert Group After

Inserts a new group **after** a selected item.

Adding and Inserting Compact Tables

It is possible to specify that all the Test Cases in a given group are displayed in the compact format (see Annex C, clause C.3 of ISO/IEC 9646-3). The following commands allow the insertion of compact groups in the test suite hierarchy:

Edit > Add Compact Group

Adds a compact group at the **next level after** the selected group.

Edit > Insert Compact Group Before

Inserts a compact group before a selected item.

Edit > Insert Compact Group After

Inserts a compact group after a selected item.

Sorting Items

The *Sort* command is applicable for any node that has dynamic subnodes.

Edit > Sort

The command sorts the sub-nodes in alphabetical order.

Cutting, Copying and Pasting Items

Editable items may be cut or copied to the clipboard. The contents of the clipboard may be pasted according to compatible classes, for example:

- Test Suite Parameters, Test Suite Constants, Test Suite Variables and Test Case Variables may be pasted into each other.
- TTCN ASP, PDU, Structured Type and CM Definitions may be pasted into each other and into TTCN ASP, PDU, Structured Type and CM Constraints (and vice versa).
- ASN.1 ASP, PDU, Structured Type and CM Definitions may be pasted into each other and into ASN.1 ASP, PDU, Structured Type and CM Constraints (and vice versa).
- Test Case, Test Step and Default Behaviours may be pasted into each other.

However, it is not possible to paste a Constraint for example as a Test Case Behaviour.

The *Paste* command is not available if the clipboard contains an object of an incompatible type to the selected object.

Note:

On the table row and text levels, no paste restrictions apply.

Edit > Cut

Deletes the selected items from the Browser and stores them in the clipboard.

Edit > Copy

Copies selected items from the Browser to the clipboard.

Edit > Paste Before

Pastes the contents of the clipboard before a selected item.

Edit > Paste After

Pastes the contents of the clipboard after a selected item.

Edit > Paste In

Pastes the contents of the clipboard to the end of a list of tables (or list of objects in a multi-object table).

Deleting Items

Editable items may be removed from the Browser.

Edit > Delete

Deletes selected items from the Browser.

> **Caution!**
>
> The TTCN Suite has no undo so this command is irreversible.

Browser Shortcut Keys for Navigation

The cursor and mouse may be used to set the input focus on an item in a Browser. However, you may find the following shortcut keys more useful:

- `<Down Arrow>` moves the input focus to the **next** item. This movement is not sensitive to the structure of a Browser.
- `<Shift+Down Arrow>` moves the input focus to the next item at the **same** level of indentation.
- `<Up Arrow>` moves the input focus to the **previous** item. This movement is not sensitive to the structure of a Browser.

- <Shift+Up Arrow> moves the input focus to the **previous** item at the **same** level of indentation.
- <Left Arrow> moves the input focus to the **previous** level of indentation.
- <Shift+Left Arrow> moves the input focus to the **previous** level of indentation and collapses that branch in a Browser.
- <Right> moves the input focus to the **first** item at the **next** level of indentation. If the next level is collapsed then it is automatically expanded.
- <Shift+Right Arrow> moves the input focus to the **last** item at the **next** level of indentation. If the next level is collapsed then it is automatically expanded.

Renaming Dynamic Items

The *Rename* command is used to give names to newly created Browser items that have the temporary name *NoName* or to rename existing Browser items. It is available in the popup menu for the item.

Rename

Renames the dynamic item. A text box with a cursor appears for the item. Type the new name and then press <Return>. It is also possible to copy, cut, paste and delete text.

- <Copy> or <L6> on a Sun keyboard copies selected text.
- <Cut> or <L10> on a Sun keyboard cuts selected text. <Ctrl+k> cuts text from the current position to the end of the line.
- <Paste> or <L8> on a Sun keyboard pastes text that is stored in the paste buffer. <Ctrl+y> pastes the most recently cut line at the current position.
- <Delete> or <Backspace> deletes one character to the left of the insertion point. <Ctrl+d> deletes one character to the right of the insertion point.

Note:

The TTCN Suite has no undo function. This means that <Undo> has no effect.

A shortcut for *Rename* is to type `<r>` without invoking the pop-up menu. The item with the input focus will then be renamed.

Opening a Table

A table is opened from a Browser when you double click on the specific item. Observe that this command has the additional effect of deselecting every item in the Browser. If this additional effect is undesirable, it can be inhibited by holding down `<Ctrl>` simultaneously.

In the case of single-object tables, point to the table name (e.g. Test Case name). To open a multiple-object table, point to a Browser item that is the title of the table (e.g. Test Case Variables) or to an object in the table (e.g. a Test Case Variable name).

In the case of collapsed items, point and double-click to expand the item (e.g. displaying all items at the next level below the pointed item). A double-click on a group table (regardless if it is collapsed or not) opens the group table. If the item pointed at is a leaf node that is not a table, the double-click will instead create a new item below it.

Alternatively, you may point and click once with the middle mouse button (i.e. set the input focus) and then press `<t>` to open the table. Note that the input focus can also be set with shortcut keys. See "Key and Button Bindings" on page 1163.

If a table has previously been opened, but it is for some reason hard to get at, it may be brought forward by selecting the name of the table in the submenu *Show Editor* of the menu *View*.

Printing a TTCN Document

To print a TTCN document:

- Select *Print* from the *File* menu.
  - The shortcut is `<Ctrl+P>`.

For more information, see "The Print Dialogs in the TTCN Suite" on page 333 in chapter 5, *Printing Documents and Diagrams*.

Requesting License Information

Help > License Info

The TTCN Suite consists of several tools and for running each tool it is required that the appropriate license is installed. The the TTCN Suite *License Information* dialog is used to list these licenses.

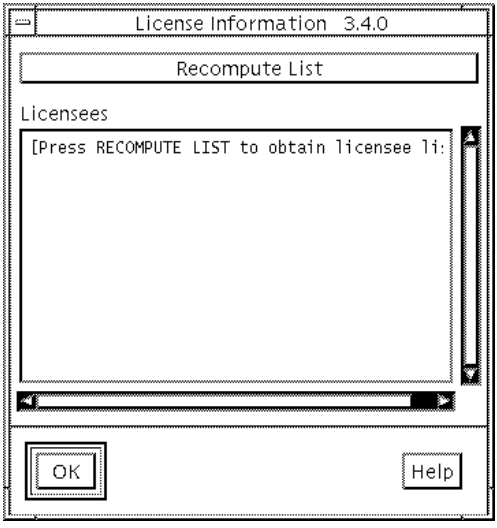


Figure 184: The Information dialog

- Clicking *Recompute List* retrieves the list of licensees and lists them in the scrollable area below.
- The *Licensees* area shows a list of licensees and the licenses available.

Using Popup Menus

Node Specific Popup Menu

This menu is shown when you point to an item in the Browser and click the right mouse button. Most of the commands are also available in the main menu bar. The commands in this menu are applied on the pointed item. Using this menu moves the input focus to the item pointed to, but does not change the selection.

Background Popup Menu

This menu is shown when you point to the background of the Browser and click the right mouse button. Using this menu does not affect the selection or the input focus.

Popup Menu Commands

The following commands in the popup menus are not available elsewhere.

Select All

Selects all items in the Browser. The same effect can be achieved by pressing `<Ctrl+/>`.

Deselect All

Deselects all selected items. The same effect can be achieved by pressing `<Ctrl+\>`.

Toggle Select

Toggles the selection state for the item. The same effect can be achieved by pressing `<Ctrl+space>`.

Toggle Select Tree

Toggles the selection state for the whole sub-tree below the item. The same effect can be achieved by pressing `<Ctrl+Shift+space>`.

Rename

Renames the item, see "Rename" on page 1120.

Using More Complex Selections

Apart from manually selecting items in the Browser, you can make more complex selections by using the Selector. These selections can then be displayed in a sub Browser.

The Selector tool is available from the Browser *Tools* menu. Like the other TTCN Suite tools, the Selector tool works on selections. The purpose of the Selector tool is to, from an existing selection, create a new selection.

With the Selector, you can make selections in three ways:

- By restricting the existing selection
- By making additions to the existing selection
- By replacing the existing selection

You can make selections by specifying restrictions on table names, restrictions on tables types and even restrictions on the contents of the tables.

Restrictions can also include the analysis status of the table, i.e. *not analyzed*, *error* and *OK*.

Selections can also be made depending on references between TTCN objects. A typical use of this feature might be to select (and perhaps later Convert to TTCN-MP) an entire Test Case, i.e. the Test Case Behaviour and all its associated declarations, constraints, Test Steps etc.

Note: UNIX only

The Selector is only available on UNIX.

Tools > Selector

Creates a new selection from a given selection. The Selector dialog is opened:

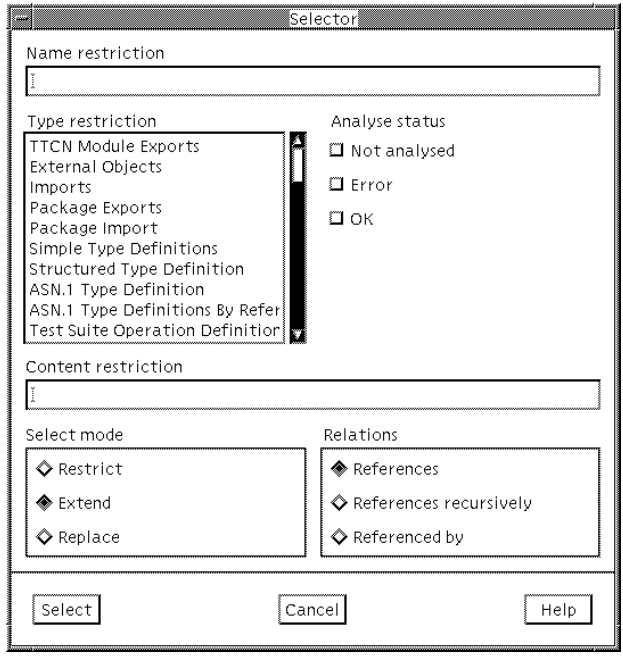


Figure 185: Selector dialog

Name Restriction

Enables the user to define the restrictions on the **identifiers** of the named TTCN tables (e.g. ASPs, PDUs, Constraints, Test Cases, Test Steps and Defaults) to be selected. This restriction is a regular expression (see "Regular Expressions" on page 1130). If the name restriction field is empty then no restrictions on the table names will apply.

Example 191 ––

The name restriction `^abc` will select all tables whose names begin with the string "abc".

––

Type Restriction

Enables the user to define the restrictions on which **kind** of TTCN tables that are to be selected. Click once to select one or more table types.

Example 192 ––

Suppose that the initial selection is the entire Dynamic Part. Choosing the table types Test Case Dynamic Behaviour and Test Step Dynamic Behaviour **together** with the name restriction `^abc` will select all test cases and test steps whose names begin with the string “abc”.

––

If no table types at all are selected then no type restrictions are assumed to apply (i.e. the same effect as selecting all table types).

Content Restriction

Enables the user to define the restrictions on the **contents** of TTCN tables that are to be selected. This restriction is a regular expression (see “Regular Expressions” on page 1130). If the content restriction field is empty then no restrictions on the contents of tables will apply.

Example 193 ––

Suppose that the initial selection is the entire TTCN document. Choosing the table type TTCN PDU Type Definition **together** with the Content restriction `^xyz` will select all TTCN PDUs which have a field beginning with the string “xyz”.

––

Select Mode

Enables the user to choose a select mode: *restrict*, *extend* or *replace*.

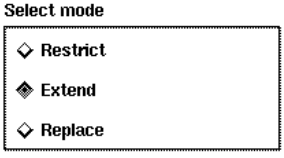


Figure 186: Selector mode

- In the *Restrict* mode the initial selection will be **reduced** to those items which fulfill the user supplied restrictions. The examples above illustrate the *Restrict* mode.

  Note that in this mode the *Relations* option is not available.

- In the *Extend* mode the initial selection will be **extended** with the items that 1) fulfill the references given by the *Relations* option and 2) fulfill the user supplied restrictions (if any).
- In the *Replace* mode items in the initial selection will be **replaced** by the items that 1) fulfill the references given by the *Relations* option and 2) fulfill the user supplied restrictions (if any).

Relations

When a TTCN document is analyzed, the TTCN Suite builds a symbol table of all the named TTCN objects that are in the TTCN document. If required, the Selector will perform a 'look-up' in this table in order to select referenced objects.

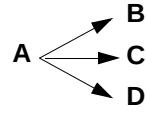
It is required that the TTCN document is analyzed before applying the Selector with the Relations option. This will ensure that the entries in the symbol table are consistent with the TTCN document on the screen.

There are three kinds of references: *References, References recursively* and *Referenced by*

- *References*

  Selects all other objects that are referenced by the initially selected objects. This new selection may be further restricted if any *Name* and/or *Type* and/or *Content* restrictions are chosen.

  In the first case, suppose that we have object A. Choosing *References* will select all the other objects, e.g. B, C and D that object A references.



Example 194 ––

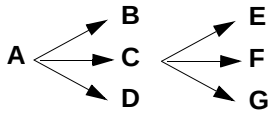
If the initial selection is a single test case and we make a type restriction of Test Step Dynamic Behaviour but no Name or Content restrictions, then choosing *Extend* and *References* will add to the current selection all the test steps (if any) that the selected test case references.

––

- *References recursively*

  Same as above except that all recursive references are also selected.

  The second case is the same as the first, except that the references are selected recursively, e.g. B, C, D, E, F and G.



Example 195 ––

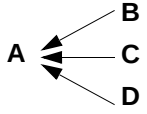
To extract an entire test, select the desired test case and apply *Extend* and *References recursively* without any other restrictions. The Selector will select the Test Case Behaviour and all its associated declarations, constraints, attached Test Steps (including any Test Steps that these Test Steps may attach), Defaults etc. The *Convert to MP* command can then be used to output this test case in TTCN-MP format.

––

- *Referenced by*

  Selects all objects that have references to the selected object. This new selection may be further restricted if any *Name* and/or *Type* and/or *Content* restrictions are chosen.

  In the third case suppose that we have object A. Choosing *Referenced by* will select all the other objects, e.g. B, C and D that reference object A.



Example 196 ––

If the initial selection is a single PDU definition, then choosing *Extend* and *Referenced by* without any other restrictions will select all the objects (e.g. constraints and behaviours) that use this PDU type.

––

Keeping and Retrieving a Selection

File > Keep Selection

Saves the current selection for later retrieval. This operation may, for example, be used when an Access application is to be applied on the TTCN document. To be able to retrieve the saved selection in another session the *Save Document* command needs to be issued after this command. For more information see "Getting Started with TTCN Access" on page 984 in chapter 21, *TTCN Access*.

File > Retrieve Selection

Retrieves the saved selection.

Searching by Using the Selector

A search pattern (regular expression) can be defined for either the complete content of a set of objects or only for the names.

A search always operates with a Browser selection as its scope. It will produce another selection as the result. By creating a sub Browser from the resulting selection a convenient environment for viewing the objects that fulfilled the search predicate is achieved.

Making Sub Browsers from Selections

Often, after making a selection, it is useful to make a sub Browser which only contains the items which are selected. This can easily be done by using the *Browser* command from the *Tools* menu of the Browser.

Example 197 ––

Applying the Selector on an entire TTCN document with the *Analyze status* option set to *error* will select all objects that have syntax and/or static semantic errors. Choosing the *Browser* command from the *Tools* menu of the Browser where the selection was applied will create a Sub-browser that only contains the error objects (including the structure needed to "glue" them together).

––

Regular Expressions

When you use the Selector or search from the Table Editor, string patterns are used for matching and searching for named objects or text strings. In UNIX terminology these patterns are called *regular expressions*. Regular expressions include symbols specifying the pattern to be searched for. Some of these symbols are:

```
\ < > ^ $ [ ] *
```

Use of these symbols in regular expressions involves a quite complicated syntax which can be very powerful. We illustrate some common uses below, but for full information we suggest that the user takes a look at the UNIX manual entry for regexp (do a *man regexp*), and look under the heading Regular Expressions.

Example 198 ----------

TTCN tables contain *fields*. Fields contain text *strings*. Text strings contain *words*.

The pattern `abc` will match all words that **include** the string “abc”, e.g. abcxyz xyzabc, xyzabcxyz, abc.

The pattern `\<abc\>` will match all **words that are exactly** “abc”. This is useful for finding TTCN identifiers embedded in strings and comments.

The pattern `^abc` will match any field that **begins** with the string “abc”, e.g. abcxyz abc, abcabc.

The pattern `abc$` will match any field that **ends** with the string “abc”, e.g. xyzabc, abc, abcabc.

The pattern `^abc$` will only match **fields containing exactly** “abc”.

The pattern `abc[1-5]$` will select all fields that end with “abc” and a single number between 1 and 5, e.g. abc1 and xyzabc4, but not abc7 or abc12.

Searching and Replacing

You specify the text to search for and the replacement text in the Table Editor, see "Searching and Replacing" on page 1176 in chapter 25, *The TTCN Table Editor (on UNIX)*. However, you can invoke the search from the Browser too and in this case, an empty Table Editor will be opened.

Edit > Find & Replace

Opens an empty Table Editor where you can specify what to search for.

Presenting Status Information

The Reporter presents status information on selected Browser items (i.e. TTCN objects) in the form of a simple list. This list can include such information as analyze status and modification dates etc. The Reporter also provides limited formatting of the list.

Reporter output may be stored to file for later use, for example printing or as input to an AWK script for further processing.

> **Note: UNIX only**
>
> The Reporter tool is only available on UNIX.

Constructing Lists with the Reporter

The Reporter tool is available from the Browser *Tools* menu.

Tools > Reporter

The Reporter tool is applied to selected items in a Browser, and presents them in the form of a simple list. Selections can either be made by using the mouse or by using the Selector tool (see "Using More Complex Selections" on page 1124).

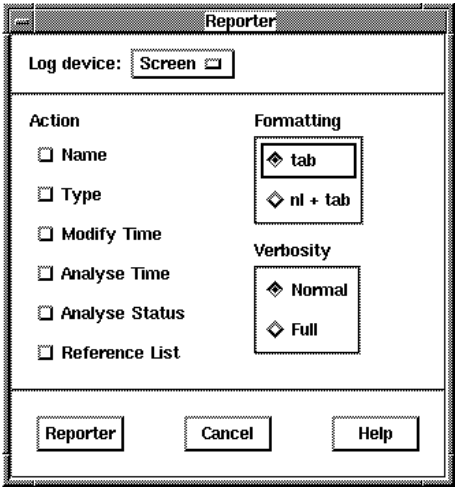


Figure 187: Reporter dialog

Log Device

Controls the log (output) device for the Reporter tool. The default value is *Screen*, but the log can be directed to a named file by choosing *File* or turn it off altogether by choosing *None*. The file extension for the reporter log is `.rpt`.

For a full description of this command see "The TTCN Suite Logs" on page 6 in chapter 1, *User Interface and Basic Operations*.

Action

The following options specify which information about each selected item shall appear in each Reporter list entry.

- *Name*

  The name of the item (e.g. ESTABLISH_CONNECTION)

- *Type*

  The type of the item (e.g. Test Step Dynamic Behaviour)

- *Modify Time*

  The date and time that the item was last modified (edited) (e.g. Thu Aug 20 13:05:55 1992).

- *Analyze Time*

  The date and time that the item was last analyzed (e.g. Thu Aug 20 13:08:35 1992).

- *Analyze Status*

  The analyze status of the item (e.g. error))

- *Reference List*

  The other objects that this object references. This information is used, for example, by the Selector when the References option is set (e.g. N_SAP:L:LT_DEFAULT:CR1:NDr:CR:CC1:NDi:CC).

Formatting

The list can be formatted using either *tab* or *nl + tab*, where *nl* stands for *new line*.

- *tab*

  In this format the components of each list entry are separated by a tab.

Example 199 ––

Part of a Reporter log showing two test steps. The list was generated using only the *Name* and *Type* option with the format option set to *tab*.

Figure 188: The TTCN Suite log for Reporter (tab)

––

- *nl + tab*

  In this format the components of each list entry each are printed on a new line. Each component, except the first component in the list entry, is preceded by a tab.

Example 200 ––

Part of a Reporter log showing two test steps. The list was generated using all the options and with the *nl + tab* format.

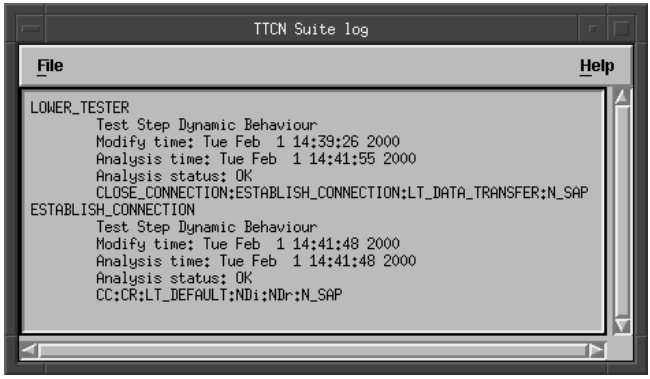


Figure 189: The TTCN Suite log for Reporter (nl+tab)

––

Verbosity

Full verbosity gives more detailed information. The following extra information is available with full verbosity:

- *Reanalysis*

  It is obvious that if the item is not OK (its analyze status), it needs to be reanalyzed. Even if the item has an OK status, the item may need to be reanalyzed since it may contain references to items which have been modified. The reporter with full verbosity will determine if the item need to be reanalyzed or not. This information is reported together with the analysis status.

- *Reference status*

  For each reference the following extra information is available with full verbosity:

  – If the reference does not exist (has been removed)
  – If it is not unique
  – If it is a named object
  – If the reference has been analyzed more recently

Revision Control

There is no integrated revision control system in the TTCN Suite. Since normal visible files are used to store the TTCN documents it is easy to integrate the TTCN Suite in a revision control system like SCCS or RCS. The `.itex` file format, being binary, is however not very suitable for that, it is better to use the TTCN-MP format to store in the revision control system.

Increased functionality is obtained if not only the TTCN-MP file is stored. The output from the Reporter contains useful information like modification dates and cross-reference lists.

By applying the following procedure whenever a check-point is desired the revision system becomes even more powerful:

1. Select the entire document.
2. Use the Convert to MP command to create a TTCN-MP file. Store it as e.g. `Test-Suite.mp`.
3. Use the Reporter, setting relevant options, to create a report. This report could now include modification date, cross-reference lists, etc. The *formatting* option *tab* should be chosen. This means that each TTCN object will occupy one line.
4. Store the log-window created by the reporter as e.g. `Test-Suite.rpt`.
5. Allow both files to be handled by the revision control system

It is now easy to obtain useful information about the difference between revisions by simply comparing the stored report files. A compare could be made using e.g. `sccs diff` (if SCCS is used).

Comparing TTCN Documents

The Compare tool is used for comparing two TTCN documents with respect to finding *similarities* that exist between them, i.e. two TTCN objects that have the same name. These objects need not necessarily be of the same type.

The Compare tool is accessed from a Browser and is applied to selections of items in that Browser.

Compare will check either only for similar names or for names **and** object type. The comparison can also include the contents of tables if required.

The Compare tool can also check the group structure of TTCN documents for similarities, i.e. it will indicate Test Group, Test Step Group and Default Group paths that are the same in both TTCN documents.

Any similarities that are found may be logged in a log file. There are three levels of log verbosity: *quiet*, *some* and *full*. Execution of the Compare tool may result in a new selection, which can be used to create a separate Browser if so wished.

One of the main uses of this tool will be to help remove conflicts from two TTCN documents prior to performing a merge operation.

Comparing by Using the Compare Tool

To use the Compare tool, you need two TTCN documents to operate on: the *destination* document and the *compare* document.

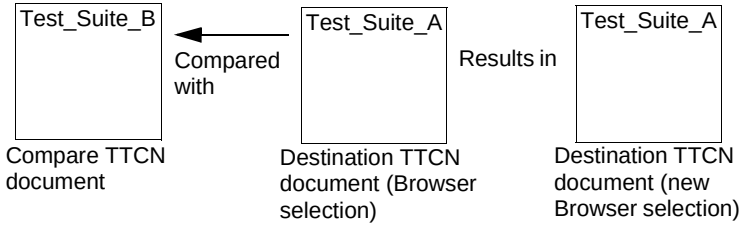


Figure 190: How Compare works

A selection from the *destination* document may be compared with the *compare* document. If any similarities are found between the two TTCN

documents this will result in the modification of the original selection in the *destination* TTCN document.

Tools > Compare

Compares a Browser selection from one TTCN document with another TTCN document with respect to similarities. Make the selection in a *destination* TTCN document Browser and call the Compare tool from the same Browser.

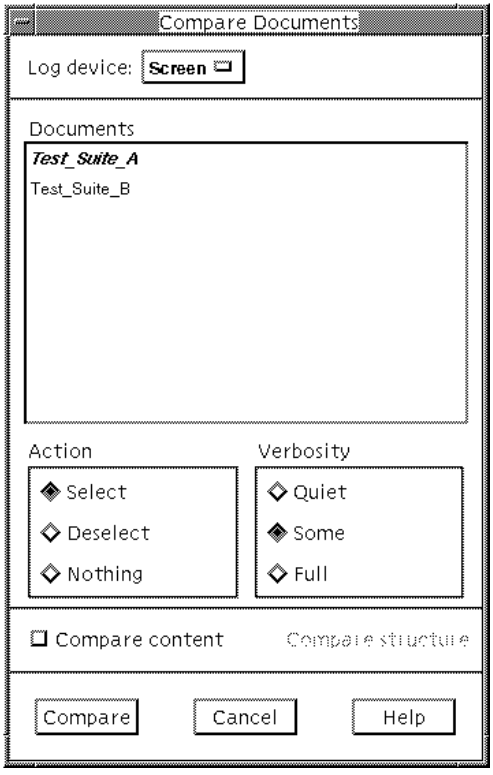


Figure 191: Compare Documents dialog

Log Device

Controls the log device for the Compare tool. The default value is *Screen*, but the log can be directed to a named file by choosing *File* or turn it off altogether by choosing *None*.

For a full description of this command see "The TTCN Suite Logs" on page 6 in chapter 1, *User Interface and Basic Operations*.

Documents

Presents a list of currently open documents. For Compare to work correctly at least two documents must be present in this list. The document name in bold italic font is the document from which the Compare tool was called, i.e. the destination document. The user must choose another document in this list, i.e. the *compare* document, in order to do a comparison.

Example 201 ––

In the dialog above the destination document is `Test_Suite_A` and hence the *compare* document may only be `Test_Suite_B`.

––

Action

Specifies the effect that the Compare operation will have on the original selection in the *destination* TTCN document.

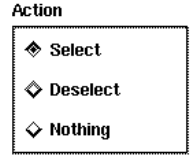


Figure 192: Compare: Action

- *Select*

  All items in the original selection that are similar to items in the compare TTCN document will **remain selected**, otherwise the items will be deselected.

In other words, the result of the Compare operation will result in a selection that shows the similarities between the destination and the *compare* TTCN documents.

- *Deselect*

  All items in the original selection that are similar to items in the compare TTCN document will be **deselected**, otherwise the items will remain selected.

  In other words, the result of the Compare operation will result in a selection that shows the differences between the TTCN documents.

- *Nothing*

  Means that the Compare operation will have no effect on the destination selection. However, the results of the Compare operation will still be recorded in the log, if this option is set.

Verbosity

Sets the verbosity level for the log file.

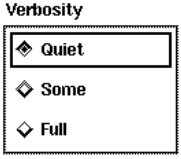


Figure 193: Compare: Verbosity

- *Quiet*

  Turns the log off. No information will be recorded in the log, even if the *Log Device* is on.

Note:

Using the *Nothing* action together with the *Quiet* option is rather meaningless!

- *Some*

  Causes the names of all identically named objects to be printed in the log.

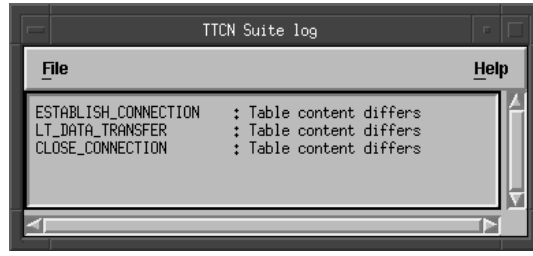


Figure 194: Compare: log (Verbosity: some)

If the *Content* switch is on, the log will also indicate that a difference has been detected.

- *Full*

  Causes the names of all identically named objects to be printed in the log. However, if the objects are of different types this will be indicated as well.

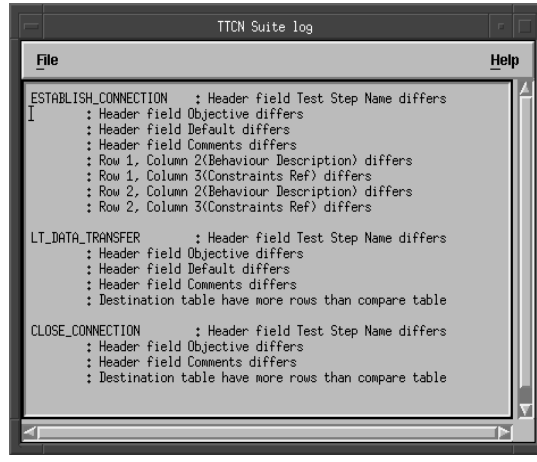


Figure 195: Compare: log (Verbosity: full)

If the *Compare Content* switch is on, the log will also indicate in detail the differences detected.

Compare Content

Extends the scope of the comparison to include the contents (down to the level of a single character) of the tables as well. Thus, two objects are considered to be identical if their names, object types and entire contents are identical. If the verbosity is full the differences are detailed.

Compare structure

Extends the scope of the comparison to include the Test Group, Test Step Group and Default Group references in the relevant behaviour tables. This option is only available when the Compare content option is chosen.

Comparing by Using itexdiff

A separate program, `itexdiff`, may also be used to quickly compare the contents of two TTCN Suite files. The tool is less powerful than the Compare tool, but it may be very useful when there is a need for quickly comparing two revisions of the same TTCN document for equality.

The program is run like:

```
itexdiff FooSuite.itex /home/user1/FooSuite.itex
```

This will test the content of the two TTCN documents for equality and report the result in a printout. To get a more elaborate compare, the `-d` switch may be added like:

```
itexdiff -d FooSuite.itex /home/user1/FooSuite.itex
```

This will give the output of running the program `diff` on the MP representation of the content of the two documents.

Running the program without any arguments will print out a terse usage message.

Using Compare Before Merging Two TTCN Documents

The Merge tool will always try to perform the merge operation, also when the two documents have conflicts. A conflict occurs if any TTCN objects in the TTCN documents have the same name.

One of the first things Merge does is to compare the two TTCN documents. Any conflicts found will be notified in the log and the Merge op-

eration will then continue. The Compare tool can be used to find and resolve the conflicts.

For a full description of this command see “Merging TTCN Documents” on page 1144.

Merging TTCN Documents

The Merge tool is used for merging one TTCN document (complete or partial) into another TTCN document.

A useful application of the Merge tool is to use the Selector tool and the Convert to MP tool to extract a complete test case (the Test Case Dynamic Behaviour together with its declarations, constraints etc.) and merge it with another TTCN document.

Preparing for a Merge

The merge tool requires that two TTCN documents are opened. We shall call them the *merge* TTCN document and the *destination* TTCN document or selection.

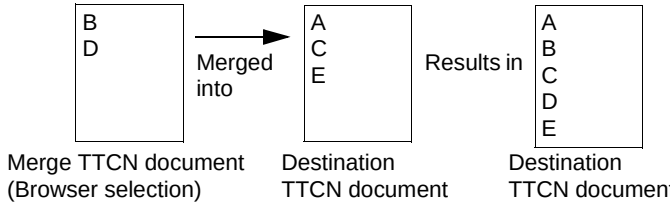


Figure 196: Shows how Merge works

The Merge tool will always try to perform the merge operation, also when the two documents have conflicts. A conflict occurs if any TTCN objects in the TTCN documents have the same name. If any name conflicts are detected, this will be logged and the conflict tables will not be merged.

One of the first things Merge does is to compare the two TTCN documents. Any conflicts found will be notified in the log and the Merge operation will then continue. The Compare tool can be used to find and resolve the conflicts.

For a full description of this command see "Comparing TTCN Documents" on page 1137.

Merging of constraint tables are handled specially. For example, a *merge* TTCN document may contain a TTCN ASP Constraint `constraint1` that refers to the type `type1`, where the `type1` Type

table is of the incompatible type TTCN PDU TypeDef. This will make the merge process insert a copy of `constraint1` as a table of type TTCN PDU Constraint instead of the original type. There are, however, limitations to this “type conversion”. The conversion will not convert an ASN.1 Constraint to a TTCN Constraint nor vice versa.

Merging Two TTCN Documents

Tools > Merge

Merges a selection into another TTCN document. Make the selection in the merge TTCN document and call the merge tool to merge the selected nodes into the destination document.

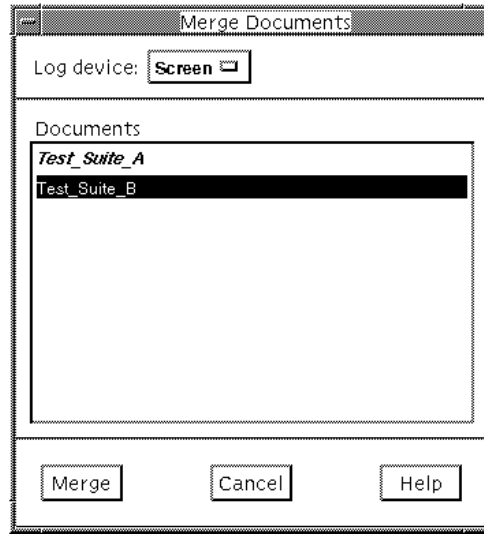


Figure 197: Merge Documents dialog

- *Log device*

  Enables the user to define the log device for the Merge tool. The default value is *Screen*, but the user can direct the log to a named file by choosing *File* or turn it off altogether by choosing *None*.

For a full description of this command see "The TTCN Suite Logs" on page 6 in chapter 1, *User Interface and Basic Operations*.

- *Documents*

  Presents a list of currently open documents. For Merge to work correctly at least two documents must be present in this list. The document name in bold italic font is the document from which the Merge tool was called, i.e. the merge document. The user must choose another document in this list, i.e. the destination document, in order to do a merge.

  In the dialog above the destination document is `Test_Suite_B` and the merge document is `Test_Suite_A`.

Creating Documents by Using the Merge Tool

The Merge Tool is not only useful when merging two documents together. It can also be used in the creation of a new document that needs to obtain some information from another document. The following procedure can be useful:

1. Create a new (empty) document.
2. Open the document containing objects that are to be copied.
3. Select the desired objects in the Browser (possibly aided by the Selector tool).
4. Merge them into the empty document.

There are also alternate ways of doing this. Copy/Paste between the documents or using the Convert to MP tool and the Convert to GR tool or the Merge from MP tool.

Merge from command line

`TTCNMerge` is a command-line tool for merging two or more TTCN documents into one.

Usage:

```
TTCNMerge <Destination TTCN Suite file> <Source file 1>
... <Source file n>
```

If `<Destination TTCN Suite file>` doesn't exist, it will be created.

Example 202 ––

```
TTCNMerge total.itex main.mp decl.mp initial.itex
```

––

The appearance order of source files in command line is significant such that files are being merged one by one in order of their appearance, and, if the table with the same name exists in several TTCN source documents, only the first appearance will be merged. All others will be skipped with warnings given.

Merging from MP Files

Use the menu choice *Merge from MP* or *Autolink Merge* for merging the content of an MP file into a TTCN document. For additional information on merging an Autolink generated MP file, see "Merging TTCN Test Suites in the TTCN Suite" on page 1398 in chapter 35, *TTCN Test Suite Generation*.

Four steps are involved in a merge from MP:

1. It first converts the MP file to a temporary GR format TTCN document, see "Importing a TTCN-MP Document" on page 1157 for details.
2. The second step is to use the Merge tool to copy all the content of the new document into the existing document, see "Merging TTCN Documents" on page 1144 for details (note the conflict resolution difference).
3. The third step is to remove the temporary document.
4. The optional fourth step is to analyze the destination document (this last step is mandatory if called under the name *AutoLink Merge*)

The Merge from MP tool requires that one TTCN document is open. We shall call it the *destination* TTCN document. This is where the merge from MP is called from.

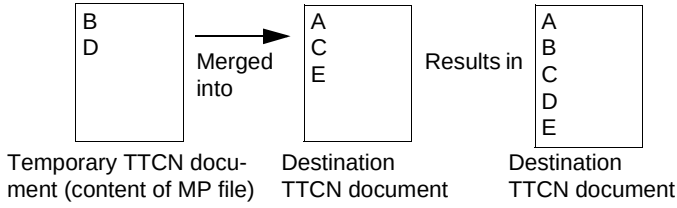


Figure 198: How a merge from MP works

Merge from MP will only work as depicted above if the two TTCN documents **do not conflict**. A conflict occurs if any TTCN object in the MP file has the same name as any TTCN object in the destination document. If such a conflict is detected, the conflicting object in the MP file will be skipped and the merge process continued.

Tools > Merge from MP

SDT Link > Autolink Merge

The menu choice *Merge from MP* in the *Tools* menu and *Autolink Merge* in the *SDT Link* menu makes it possible to merge the content of an MP file into the current TTCN document.

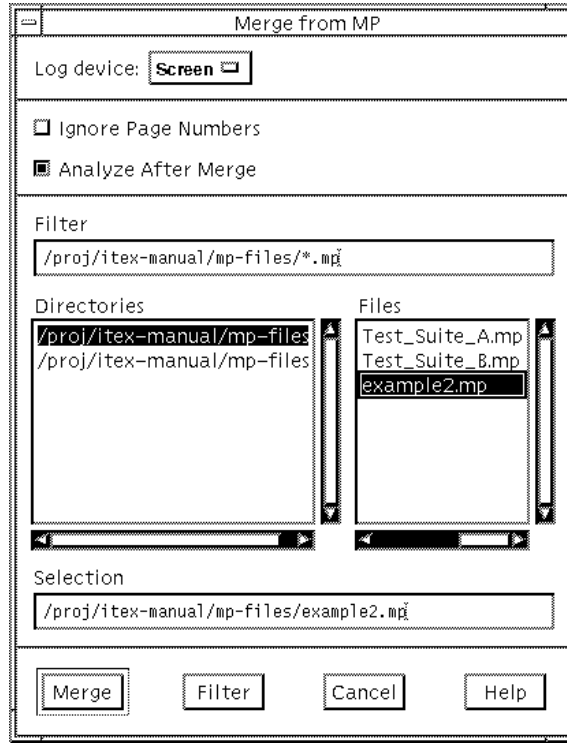


Figure 199: The Merge from MP dialog

- *Log device*

  Enables you to define the log device for the merge from MP. The default value is *Screen*, but the user can direct the log to a named file by choosing *File* or turn it off altogether by choosing *None*.

  For a full description of this command see "The TTCN Suite Logs" on page 6 in chapter 1, *User Interface and Basic Operations*.

- *Ignore Page Numbers*
  Ignore any page number information in the MP file if set.

- *Analyze After Merge*

  Runs the Analyzer on the whole destination document after successful merge if set. This option is not possible to disable when using *AutoLink Merge* and so is not displayed in that dialog.

- *Filter*

  Sets the filter for the files that will be displayed in the *Files* list. By convention TTCN-MP files are given the suffix `.mp`.

  Example 203 ––

  The name filter ***** `.mp` will cause only those files whose names end with `.mp` to be displayed, e.g. `example.mp`

  ––

- *Directories*

  Lists the directories in the current file system directory. To change directory, point to the required directory name and double-click.

- *Files*

  Lists the files (if any) that are in the current directory and which match the filter.

- *Selection*

  Displays the selected document.

- *Merge*

  Terminates the dialog and executes the merge from MP.

- *Filter*

  Updates the *Files* list according to the current filter.

Exporting a TTCN Document to TTCN-MP

TTCN-MP – the textual notation (*m*achine *p*rocessable) – can be used when you want to import a TTCN document into a non-TTCN Suite tool or make backups.

By using the menu command *Convert to MP*, you can convert a TTCN-GR document to TTCN-MP. However, note that it is also possible to simply use *Save Document As* for changing the format. The difference between the two menu commands is that *Convert to MP* gives the option to save as TTCN Suite MP or standard MP, while *Save As* produces TTCN Suite MP. How this differs from the standard MP format is described in "The TTCN Suite MP format" on page 1154.

There are no restrictions on the analysis state of the TTCN document with regards to its correctness, i.e. it will generate TTCN-MP even for TTCN documents that are neither complete nor error-free. This implies that the generated MP file may be, but is not necessarily, conformant to the standardized TTCN-MP format.

However, to ensure that a TTCN document that is output in TTCN-MP by the TTCN Suite can be read by a non-TTCN Suite tool, it is necessary to make sure that the TTCN document is correctly analyzed, that the test suite overview have been recently updated. You also have to select everything in the document before convert to MP. The TTCN Suite format option in the dialog will in most such cases have to be non-selected.

Convert to MP, invoked from the Browser, works on Browser selections, i.e. it is possible to output selected items (e.g. a single test case or just the constraints). Convert to MP will add the necessary keywords to the MP file to make it into a TTCN-MP that can be converted back into GR format.

An optional but non-standard TTCN-MP for compact tables and ASN.1 references is supported. Additional fields in dynamic tables (fields which are transferred to test suite overview tables) are also supported. This format is the recommended format for transferring TTCN document between the TTCN Suite instances, and also for making backups of the TTCN documents.

Convert to MP is accessible from the Browser *File* menu and from the Organizer *Generate* menu.

Converting to TTCN-MP in the TTCN Suite

The items selected in the Browser will be converted to TTCN-MP. Selections can be arbitrary – the TTCN Suite will add the necessary keywords to the MP file to make it into a TTCN-MP that can be converted back into GR format.

To convert an entire TTCN document, do a *Select All* and then select *Convert to MP* from the *File* menu, or invoke this operation from the Organizer. For more information, see "Convert to MP (TTCN)" on page 133 in chapter 2, *The Organizer*.

For example, to convert a complete test case, use the Selector, together with the *References recursively* option and then apply *Convert to MP* on the resulting selection.

File > Convert to MP

Presents a dialog in which you can convert the selected items to MP.

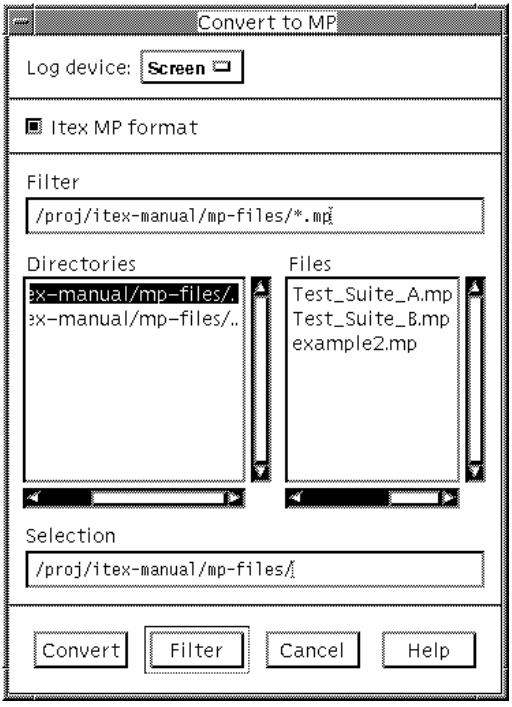


Figure 200: Convert to MP file dialog

Log Device

Controls the log device for the Convert to MP tool. The default value is *Screen*, but the log can be directed to a named file by choosing *File* or disabled by choosing *None*.

For a full description of this command see "The TTCN Suite Logs" on page 6 in chapter 1, *User Interface and Basic Operations*.

The TTCN Suite MP format

Enables the user to convert a TTCN document to a non-standard MP with the following extra information:

- *Compact tables*

Is intended for TTCN documents where the Test Case Dynamic Behaviours (i.e. compact test cases) table has been used. Check this option to generate an MP file that contains additional keywords that specifically identify compact tables. If this option is not used the TTCN Suite will output a standard TTCN-MP file where each test case in the compact test cases table is represented as a single test case (i.e. the information about the compact table is lost).

- *Additional fields in dynamic tables*

  If the *ITEX MP format* option is used, the TTCN Suite will convert the additional Selection Ref and Description fields in the behaviour tables.

- *ASN.1 reference tables*

  The additional field in ASN.1 type by reference and Test Suite Constant Declarations By Reference tables will be converted if this option is checked. These additional fields contain the definition of the ASN.1 type and value.

- *Import tables*

  The additional column fields in Import tables will also be converted if this option is checked.

Filter

Sets the filter for the files that will be displayed in the *Files* list. By convention TTCN-MP files are given the suffix `.mp`.

Example 204 ––

The name filter ***** `.mp` will cause only those files whose names end with `.mp` to be displayed, e.g. `example.mp`

––

Directories

Lists the directories in the current file system directory. To change directory, point to the required directory name and double-click.

Files

Lists the TTCN Suite TTCN-MP files (if any) that are in the current directory and which match the filter.

Selection

Displays the selected document.

Filter

Updates the *Files* list according to the filter.

Exporting by Using itex2mp

A separate program, `itex2mp`, may be used to quickly export the content of a TTCN Suite file. The generated MP format includes the TTCN Suite-specific extensions.

The program is run like:

```
itex2mp FooSuite.itex FooSuite.mp
```

This will export the content of the TTCN document into the specified MP file. You may notice that the program has the “style” of a specialized `cp` program.

Running the program without any arguments will print out a terse usage message.

Importing a TTCN-MP Document

By using the menu choice *Convert to GR* in the Organizer, you can convert a TTCN-MP document to TTCN-GR and open it in the TTCN Suite. However, note that it is also possible to simply open an MP document.

The Convert to GR tool is flexible, it will read any TTCN-MP file that has been successfully generated by the Convert to MP tool, even if the MP file only contain parts of a test suite. Convert to GR will also read TTCN-MP from other products which support TTCN assuming that this TTCN-MP follows the ISO standard.

For a full EBNF supported by the Convert to GR tool, see "The TTCN-MP Syntax Productions in BNF" on page 1574 in chapter 38, *Languages Supported in the TTCN Suite*.

An optional but non-standard TTCN-MP for converting TTCN documents that use compact tables and ASN.1 references with an explicit definition, is also supported.

Converting a TTCN Document

For a full description of this command see "Convert to GR (TTCN)" on page 136 in chapter 2, *The Organizer*.

MP File Format Problem

When converting certain TTCN documents, a problem with transferring the information (e.g. Description) in the Overview tables to the tables in Dynamic Part, may occur.

The TTCN standard allow path specifications to optionally include the document name first. This has the unfortunate effect that if the TTCN document contains a top level group with the same name as the TTCN document there is, in general, no way of knowing if the first part of the paths is a group identifier or the TTCN document identifier.

The TTCN Suite assumes that if the first part of the path is equal to the document name, it is the optional document name and, when converting, strips it away. When exporting, the document name is always added to the front of all paths. That way the TTCN Suite is always able to convert the MP files it exports. Note that ITEX 2.0 did not add the docu-

ment name at export, and therefore the problem described here may also apply when converting MP files exported by ITEX 2.0.

If the TTCN document contains top level group identifiers equal to the document name, and the TTCN Suite is unable to resolve the paths, temporarily change the document name in the MP file and change it back once inside the TTCN Suite.

Converting Fields Containing the Dollar Character

Normally the Convert to GR tool ignores the content of the fields of tables in the MP file but, to allow converting of non-BoundedFreeText fields (e.g. TS_VarValue field) with embedded dollar characters (e.g. “$”) the Convert to GR tool has been enhanced with a simple syntax checker of the fields. This has the implication that it is no longer possible to convert TTCN documents containing unmatched single or double quotes in non-BoundedFreeText fields, i.e. Convert to GR is less tolerant about syntax errors in the fields.

Generating the Test Suite Overview Tables

The TTCN Suite generates the contents of the Test Suite Overview tables (i.e. The Test Suite Structure, Test Case Index, Test Step Index and Default Index tables) and the TTCN Module Overview (i.e. The TTCN Module Structure and the index tables).

These tables are **not** editable. The extra information that they require (such as references to selection expressions and descriptions) are entered at the corresponding table rather than in the Test Suite Overview tables. This ensures consistency between the TTCN document and the contents of the Test Suite Overview.

Selection Expressions

The Test Case Dynamic Behaviour table displayed by the TTCN Table Editor has an additional field in the table header called: *Selection Ref.*

This field is used to hold the reference to a Selection Expression (if any) declared in the Test Case Selection Expression Definitions table. This reference will then appear in the *Selection Ref* column of the Test Case Index when the index is generated using the Generate tool.

If the Selection Reference needs to be edited this must be done in the relevant Test Case table and not in the Test Case Index table (which cannot be edited). The Test Suite Overview (or the TTCN Module Overview) must be re-generated in order to update any changes.

The additional *Selection Ref* field in the Test Case header is **not** printed in the TTCN-GR form. It is however exported in the TTCN Suite specific MP format.

Test Case, Test Step and Default Descriptions

The Test Case, the Test Step and the Default Dynamic Behaviour tables displayed by the TTCN Table Editor have an additional field in the table header called *Description*.

The *Description* field is used to hold the description of the behaviour table, which in many cases may simply be a copy or a shortened form of the Test Purpose or Test Step or Default Objective. This description will then appear in the Description column of the relevant index when it is generated using the Generate Overview tool.

If the Description needs to be edited this must be done in the relevant behaviour table and not in the index tables (which cannot be edited). The Test Suite Overview (or the TTCN Module Overview) must be re-generated in order to update any changes.

The additional Description field in the Test Case, Test Step and Default headers is **not** printed in the TTCN-GR form. It is however exported in the TTCN Suite specific MP format.

The Group Table

The TTCN Browser supports additional non-standard tables for groups. These tables are: the *Test Group* table, the *Test Step Group* table and the *Default Group* table. **None** of these tables are printed in the TTCN-GR form.

The Test Group table has two fields in its header that are used to hold information for the Test Suite Structure table. These fields are: *Selection Ref* and *Test Group Objective*.

The *Selection Ref* field is used to hold a reference to a Selection Expression (if any) declared in the Test Case Selection Expression Definitions table. This reference will then appear in the Selection Ref column of the Test Suite Structure when it is generated using the Generate tool.

The *Test Group Objective* field is used to hold the description of the Test Group. This description will then appear in the *Test Group Objective* column of the Test Suite Structure when it is generated using the Generate tool.

If either the Selection Reference or the Test Group Objective needs to be edited this must be done in the relevant Test Group table and not in the Test Suite Structure table (which cannot be edited). The Test Suite Overview must be re-generated in order to update any changes.

All of these fields are exported in the TTCN Suite specific MP format.

Generation of the Test Suite Overview and Indices

The Generate tool is available from the *Tools* menu in the Browser and the *Tools* menu in the Table Editor. Unlike other tools it does **not** require a selection: it always applies to the entire TTCN document and its action is the same whether it is invoked from a Browser or from a Table Editor.

Tools > Generate SO

Generates the Test Suite Overview (or the TTCN Module Overview) tables (i.e. The Test Suite Structure, TTCN Module Structure, Test Case Index, Test Step Index and Default Index tables). It makes use of the additional Group tables and the additional Selection Ref and Description fields in the behaviour tables.

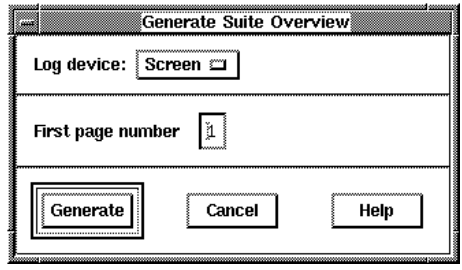


Figure 201: Generate Suite Overview dialog

- The *Log Device* option controls the log device for the Generate tool. The default value is *Screen*, but the log can be directed to a named file by choosing *File* or turn it off altogether by choosing *None*.

  For a full description of this command see "The TTCN Suite Logs" on page 6 in chapter 1, *User Interface and Basic Operations*.

- The *First Page Number* option enables setting the first page number of the TTCN document. The default value is 1.

Crash Recovery

In the event of a TTCN Suite crash, it is possible to recover up-to-the-minute changes. The auto-save content is stored in two “comma files”: `filename,s` and `filename,t`, where `filename` consists of the name of the file they are associated with and in some cases a '#' character and an integer number. The '#' character and the integer number are added when a file with the same name already exist. This can happens when a crash has occurred previously or when the Suite with the same filename is opened in another TTCN Suite Browser. The integer number is increased until a unique filename can be created. The files can be found in the designated temporary directory, usually `/var/tmp`. To be able to open those files, you need to concatenate them into a `.itex` format file. This is accomplished with the `iconcat` program.

The program is run like:

```
iconcat /var/tmp/FooSuite FooSuite-saved.itex
```

This will concatenate the content of the TTCN Suite files `/var/tmp/FooSuite,s` and `/var/tmp/FooSuite,t` into the `FooSuite-saved.itex` ITEX file. Note that the program will ignore a `/var/tmp/FooSuite` file, despite the argument syntax.

You may notice that the program has the “style” of a specialized `cp` program. Running the program without any arguments will print out a terse usage message.

Key and Button Bindings

Key	Action
Ctrl+Shift+space	Toggles the select status of all nodes in the subtree.
Ctrl+space	Toggles the select status of the focused node.
Shift+space	Set the select status of all nodes in the subtree. Remove select status on all others.
Space	Set the select status of the focused node. Remove select status on all others.
t	Open an editor on the focused node.
osfHelp	Get help on Browser.
c	Hide subtree.
e	Make children of the focused node visible.
Shift+e	Make whole subtree visible.
r	Rename the focused node.
Ctrl+c	Copy the focused node (for later paste).
Ctrl+x	Cut the focused node (for later paste).
Ctrl+v	Paste the content of the paste buffer before/above the focused node.
Ctrl+Shift+osfInsert	Add a new child to the focused node. It is placed last in the child list.
Ctrl+osfInsert	Add a new sibling before/above the focused node.
Ctrl+Meta+osfInsert	Add a new sibling after/below the focused node.
Ctrl+Shift+G	Add a new group child to the focused node. It is placed last in the child list.
Ctrl+g	Add a new group sibling before/above the focused node.
Ctrl+Meta+g	Add a new group sibling after/below the focused node.

Key	Action
F2	Move down in tree such that if we start on root node we come to “Test Cases”.
Ctrl+/	Select all nodes in browser.
Ctrl+\	Deselect all nodes in browser.
osfUp	Move focus to previous visible node.
osfDown	Move focus to next visible node.
osfLeft	Move focus to parent of the focused node.
osfRight	If the children of the focused node are hidden, make them visible. Then move focus to first child.
Shift+osfUp	Move focus to previous node at same level.
Shift+osfDown	Move focus to next node at same level.
Shift+osfLeft	Move focus to parent of the focused node, and hide the subtree.
Shift+osfRight	If the children of the focused node are hidden, make them visible. Then move focus to last child.
osfBeginLine	Move to first visible node of browser.
osfEndLine	Move to last visible node of browser.
Shift+F10	Show node popup menu.
Ctrl+Z	Run “Analyze & Stop” on page 1191 in chapter 26, *Analyzing TTCN Documents (on UNIX)*

Mouse button (OSF Names)	Action
`Button1`	Set focus to the node and set selection status of the node. Remove select status on all others.
`Ctrl+Button1`	Set focus to the node and toggle selection status of the node.
`Shift+Button1`	Set focus to the node and set selection status of the subtree. Remove select status on all others.
`Ctrl+Shift+Button1`	Set focus to the node and toggle selection status of the subtree.
`Button1` twice	Set focus to the node and start an editor on that node if that is at all reasonable. If the node is a static node, make its children visible or hidden. Remove select status on all nodes.
`Ctrl+Button1` twice	Set focus to the node and start an editor on that node.
`Button2`	Move focus to the node.
`Button3`	Show popup menu.

Chapter

25

The TTCN Table Editor (on UNIX)

This chapter contains reference information about the Table Editor, used for editing various TTCN tables.

The functionality of the Table Editor, the menus, windows and quick-buttons are described in this chapter. The TTCN Browser is described in chapter 24, *The TTCN Browser (on UNIX)*. How to analyze and find tables is described in chapter 26, *Analyzing TTCN Documents (on UNIX)*.

For an overview of the TTCN Suite, see chapter 3, *Introduction to the TTCN Suite (on UNIX), in the TTCN Suite 6.2 Getting Started*.

> **Note: UNIX version**
>
> **This is the UNIX version of the chapter. The corresponding Windows chapter is chapter 30, *Editing TTCN Documents (in Windows)*.**

Introduction to the TTCN Table Editor

The Table Editor is used for editing the various TTCN tables. It is possible to have any number of tables open for editing at any given time.

The TTCN Suite supports all of the standardized, non-compact tables that are defined in ISO/IEC 9646-3. The compact Test Case Dynamic Behaviour table (ISO/IEC 9646-3, clause C.3) and the concurrent TTCN tables are supported, as well as the modular TTCN tables to be added in the upcoming new TTCN version.

The compact tables for constraints (also in Annex C of ISO/IEC 9646-3) are not supported.

On screen, the TTCN Suite presents all tables exactly as defined in ISO/IEC 9646-3, apart from some minor additions to the regular dynamic behaviour tables (for test cases, test steps and defaults), to the compact test case dynamic behaviour table, to the ASN.1 type by reference tables, to the test suite constant declarations by reference, and to the import tables.

These changes have been introduced to aid the automatic generation of the TTCN document overview and index tables. For more information, see "Generating the Test Suite Overview Tables" on page 1159 in chapter 24, *The TTCN Browser (on UNIX)*. However, in the printed GR form the TTCN document is presented according to the ISO standard (that is, the additions are not present in the printed version).

As a visual aid, dotted lines separate rows in a table. Again, this is not present in the printed GR form.

The TTCN Suite always displays the optional comments column and comments footer of a table, even if they are empty.

You can use both menu choices and keyboard shortcuts for editing tables. It is also possible to search and replace text patterns. By providing a Data Dictionary, the Table Editor supports semi-automatic generation of send and receive statement lines. In the Data Dictionary, you select the components in a send or receive statement from lists of PCOs, types and constraints. The TTCN Suite prevents you from selecting incompatible items.

The Table Editor also shows the analysis status of the table contents.

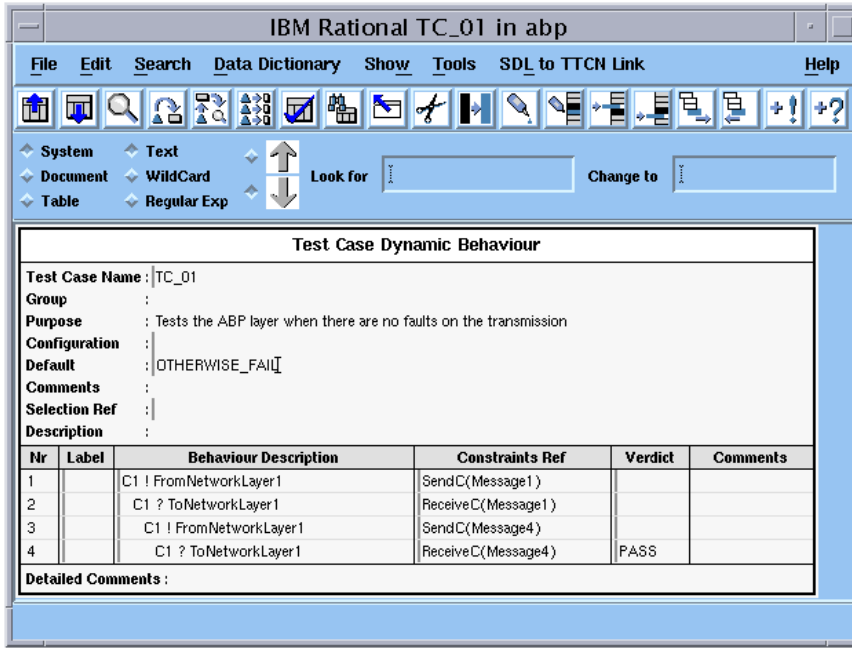


Figure 202: A TTCN Table Editor for a test case

Opening a Table

Tables are opened from a Browser when you double-click on a selected table name, or with the popup menu. For more detailed information see "Opening a Table" on page 1121 in chapter 24, *The TTCN Browser (on UNIX)*.

Renaming a Table

A table can be renamed either by using the *Rename* command in the Browser (for more information see "Renaming Dynamic Items" on page 1120 in chapter 24, *The TTCN Browser (on UNIX)*) or by editing the table name in the table itself. This choice also exists for objects in multiple-object tables.

Navigating and Editing Text in a Table

The Table Editor facilitates fast and efficient editing. The input focus determines which field is open for editing at any given time.

Setting the Input Focus

The input focus is automatically set on the name field of the table when it is opened. Reset the input focus by using the following shortcuts:

To move to:	Use this key combination:
Next field	<Tab>
Previous field	<Shift+Tab>
Next part of the table	<Ctrl+Tab>
Next field in same column	<Ctrl+down arrow>
Previous field in same column	<Ctrl+up arrow>
Field on the left	<Ctrl+left arrow>
Field on the right	<Ctrl+right arrow>

You can also use the mouse for setting the input focus. However, note that a middle-click only moves the focus to that field but it is not possible to edit text. To do this, you have to left-click in the field.

Editing Text in a Table

To edit the text in a field, you set the input focus and start typing. The table fields will expand automatically to accommodate the text and you can add a line break by simply pressing <Return>. To move the insertion point in the same field, you can either use the mouse or the following shortcuts:

To move the insertion point:	Use this key combination
Down	<Ctrl+N>, <down arrow>
Up	<Ctrl+P>, <up arrow>
Left	<Ctrl+B>, <left arrow>

To move the insertion point:	Use this key combination
Right	<Ctrl+F>, <right arrow>
To the beginning of the line	<Ctrl+A>
To the end of the line	<Ctrl+E>
Left and extending selection	<Shift+left arrow>
Right and extending selection	<Shift+right arrow>

If you initially type with very high speed, race problems may arise and you will be notified with a beep. This means that the text typed in will not appear correct in the buffer. If this happens often, you should make a brief pause between typing the first and consecutive characters.

Cut, Copy and Paste Text

To cut, copy and paste text in the fields, you can use the following shortcuts. Note that you do not use the same commands or shortcuts as when editing entire rows in a table.

To do this:	Use this key combination:
Delete one character forward	<Ctrl+D>
Cut selected text	<Cut>, <Shift+Delete>
Cut text to the end of the line	<Ctrl+K>
Copy selected text	<Copy>, <Ctrl+Insert>[a]
Paste cut or copied text	<Paste>, <Shift+Insert>
Paste the most recently cut line	<Ctrl+Y>

a. <Ctrl+Shift+Insert> also works.

Note:

The paste buffer for text is not the same as the one used for entire rows in the body of a table.

Caution!

The TTCN Suite has no undo function. This means that `<Undo>` has no effect.

Editing Rows in a Table

Rows can be added, deleted, copied, etc. in the bodies of all TTCN tables that contain more than one column. The ASN.1 tables only have a single column with a single row and therefore adding, deleting and copying rows is not applicable. However, you can still copy and paste the contents of these tables as text.

You cannot add or remove rows in table headers and footers, as the formats of these parts of a table are defined by the TTCN standard.

Selecting Rows in a Table

In the dynamic behaviour tables, you select a single row by clicking the row number.

You select a *behaviour tree* of rows by `<Ctrl+Shift>`-clicking.

You select rows in the body of a table by `<Ctrl>`-clicking. Selecting a row does not deselect other selected rows.

To deselect a row, you select it again.

> **Note:**
> Setting the input focus will also deselect all selected rows.

Cutting, Copying and Pasting Rows

Rows in a table may be cut, copied and pasted. This not only possible among tables of the same or similar type, but also across different types of tables.

Edit > Cut

Removes selected row or rows from the table and stores them in the clipboard.

Edit > Copy

Copies selected rows to the paste buffer. Any number of rows may be selected.

Note:

The paste buffer used for entire rows in the body of a table, is not the same as the one used for text.

Edit > Delete

Deletes the selected row or rows.

Caution!

The TTCN Suite has no undo so this command is irreversible.

Inserting Rows

Edit > Insert Row

Inserts a new row before a selected row. Only one row may be selected.

Edit > Insert Row After

Inserts a new row after a selected row. Only one row may be selected.

Edit > Insert Tree Header

Inserts a new Tree Header row before a selected row. Only one row may be selected. This command only works in dynamic behaviour tables.

Edit > Insert Tree Header After

Inserts a new Tree Header row after a selected row. Only one row may be selected. This command only works in dynamic behaviour tables.

The <Ins> Key

The `<Ins>` key (on the right-hand keypad) can be used for inserting rows. If you press this key while input focus is in the header or footer of a table, a new row is appended after the **last** row in the body of the table. If the input focus is set on a field in the body of the table, `<Ins>` will

insert a new row **after** the field that has the input focus. The input focus will be transferred to the corresponding field in the new row.

Indenting Rows in Behaviour Descriptions

TTCN *behaviour* tables include a number of commands and shortcut keys that simplify indenting behaviour lines.

> **Caution!**
>
> Do not use the <Tab> character to represent indentation in behaviour trees.

Edit > Increase Indent

Increases the indentation of selected behaviour lines by one position. This command can only be used to indent behaviour lines.

- Keypad <+>
  Pressing <+> (*Plus*) on the right-hand keypad (the normal <+> key does not have this effect) will indent the field that currently has the input focus.
- <Ctrl> + Keypad <+>
  Pressing <Ctrl> together with the *keypad* <+> key will cause **selected** behaviour lines to be indented one position, i.e. it has the same effect as the *Increase Indent* command.
- <Shift> + Keypad <+>
  Pressing <Shift> together with the *keypad* <+> key will cause an entire **branch** in the behaviour tree to be indented one position.

Edit > Decrease Indent

Decreases the indentation (i.e. *undent*) of the selected rows by one position. This command can only be used on behaviour lines.

- Keypad <->
  Pressing <-> (*Minus*) on the right-hand keypad (the normal <-> key does not have this effect) will undent the field that currently has the input focus.

- <Ctrl> + Keypad <->
  Pressing <Ctrl> together with the *keypad* <-> key will cause **selected** behaviour lines to be undented one position, i.e. it has the same effect as the *Decrease Indent* command.

- <Shift> + Keypad <->
  Pressing <Shift> together with the keypad <-> key will cause an entire **branch** in the behaviour tree to be undented one position.

Selecting Branches in Behaviour Descriptions

TTCN *behaviour* tables include the following short-cut key that simplifies the task of selecting behaviour lines.

- <Shift+Ctrl> + Left Mouse Button
  Pressing <Shift+Ctrl> together with the left mouse button while the cursor is pointing to a behaviour line, will select that behaviour line and all the subsequent behaviour lines that are in the same branch of the behaviour tree.

Showing the Indentation Level

Show > Show Indent

Causes the indentation level of behaviour lines to be displayed in the line number column of behaviour tables. Choosing this command again will cause the display to revert to line numbering.

Searching and Replacing

From the Table Editor, you can search for and replace text, either on system, document or table level. You specify the search in the search bar of the Table Editor and start the search (and replace) from menu choices in the *Search* menu.

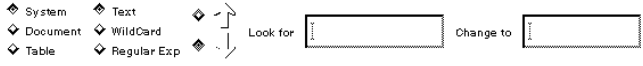


Figure 203: The TTCN Table Editor search bar

Specifying Search and Replace Settings

Delimiting the Scope of Search

You can search for text on three levels:

- *System* means all documents of the associated document – except for any Flat View document – or only the associated document if it is a Flat View.
- *Document* means the associated document only.
- *Table* means the current table only.

Selecting the Type of Search

There are three types of search patterns:

- *Text* means ordinary literal strings.
- *Wildcard* means a pattern with * and ?.
- *Regular Exp* means regular expression.

  See also "Regular Expressions" on page 1130 in chapter 24, *The TTCN Browser (on UNIX)*.

Selecting the Search Direction

The search direction can be forward (down arrow) or backward (up arrow).

Specifying the Text to Search For

You specify the text to search for in the *Look for* field. If you search for text using a regular expression, you will be notified if you try to use an incorrect regular expression.

Specifying the Replacement Text

You specify the replacement text (or replacement pattern, if the type of search is regular expression) in the *Change to* field.

If you search and replace using regular expressions, the replacement pattern must be compatible with the search pattern. If it is not, you will be notified.

Something important to note about regular expressions, is that the actual replacement text is a result of a combination of the search pattern, the match found and the replacement pattern. This means that the actual replacement text may not literally match the text in the *Change to* field. However, the actual replacement text will be presented in the status bar when you start searching and a match is found.

Starting the Search and Replace

Search > Search

Starts the search from the current position. If you search in the current table, selected text will be skipped. If the Table Editor is empty, the search will start from the beginning or end of the document or system, depending on the scope and direction that you have specified.

Search > Replace

Replaces currently selected text with the replacement text. If you searched using regular expressions, the replacement text will be presented in the status bar when a match is found.

You can also replace manually selected text. This only requires that *Text* or *Wildcard* is selected and that the *Change to* field contains some text. Note that you cannot manually make an empty selection, but it is possible to search and find an empty match that you can replace.

Search > Proceed

Searches for and replaces text sequentially. If text is selected, it will be replaced and after that the search will proceed.

Search > Replace All

Replaces selected text and then continues the search and replace from the current position. If the Table Editor is empty, the search will start from the beginning or end of the document or system, depending on the scope and direction that you have specified. The number of replacements will be presented in the status bar.

Exporting and Importing Objects

Generate Exports and *Generate Import* assist in the use of the Modular TTCN feature of exporting and importing objects to and from other documents.

Tools > Generate Exports

Fills the TTCN Exports table with rows that provide access to the objects of the current document. Only rows that are not already present will be added (at the end of the table), so it is supported to repeat this operation at a later time to detect if any object has been added since the last time. *Generate Exports* is only available when the Table Editor contains a TTCN Exports table.

Tools > Generate Import

Fills the Import table with rows that enables access to the objects of the document to import from. Only rows that are not already present will be added (at the end of the table), so it is supported to repeat this operation at a later time to detect if any object has been added since the last time. *Generate Import* is, naturally, only available when the Table Editor contains an Import table.

Browsing in the Table Editor

With the *Previous Table* and the *Next Table* commands, you can easily browse the contents of the TTCN document.

File > Previous Table

Replaces the table shown in the Table Editor with the previous table in the document. To open the table five tables before the current, use `<Ctrl>` + *Previous Table*.

File > Next Table

Replaces the table shown in the Table Editor with the next table in the document. To open the table five tables after the current, use `<Ctrl>` + *Next Table*.

See also "Context Sensitive Popup Menu" on page 1185 and "Finding Tables by Name" on page 1207 in chapter 26, *Analyzing TTCN Documents (on UNIX)*.

Generating Behaviour Statements

In TTCN *behaviour* tables it is possible to get lists of user defined objects (e.g. constraints, timers etc.). Through these lists, you can generate a statement.

When you invoke the following commands, a row in the table should have the input focus.

Data Dictionary > Add Send Statement

Generate a Send statement by selecting a PCO, a type and a constraint.

This functionality requires that the TTCN document is analyzed. The PCOs, types and constraints with major reference problems or missing type references will not be presented in these lists.

Each list is split in two parts. The upper part contains the items corresponding to the selected items in the other lists. The lower part contains the rest of the items.

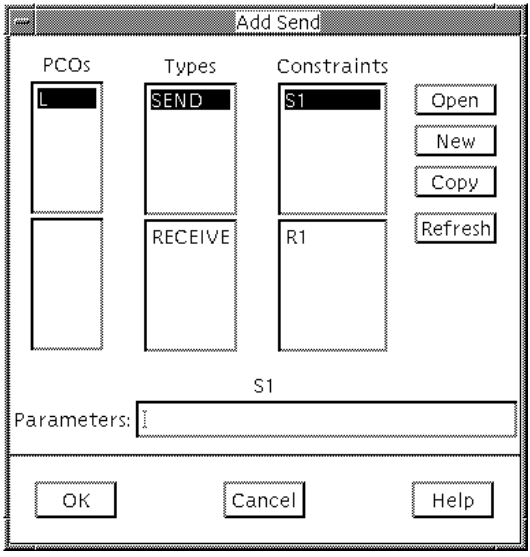


Figure 204: Add Send statement dialog (receive is analogous)

- *PCOs*

  Select a PCO or CP in this list and the *Types* and *Constraints* lists will be split accordingly.

- *Types*

  Select an ASP, PDU or CM type in this list and the corresponding constraints and *PCOs* lists will be split accordingly.

- *Constraints*

  Select a constraint in this list and the corresponding Types and PCOs will be split accordingly.

- *Open*

  The selected constraint will be opened.

- *New*

  A new constraint of the selected type is created. The new constraint will be opened.

- *Copy*

  A new constraint which is a copy of the selected constraint is created. The new constraint will be opened.

- *Refresh*

  Recompute the contents of the lists and re-display the dialog. The selection will be preserved.

- *Parameters*

  If the selected constraint has a parameter list, an actual parameter list is specified here.

- *OK*

  A Send statement is generated with the selected PCO (or CP), type and constraint.

Data Dictionary > Add Receive Statement

Generate a Receive statement by selecting a PCO, a type and a constraint.

A dialog which is analogous with *Add Send Statement* will be displayed.

Data Dictionary > Add StartTimer Statement

Generate a StartTimer statement by selecting a timer.

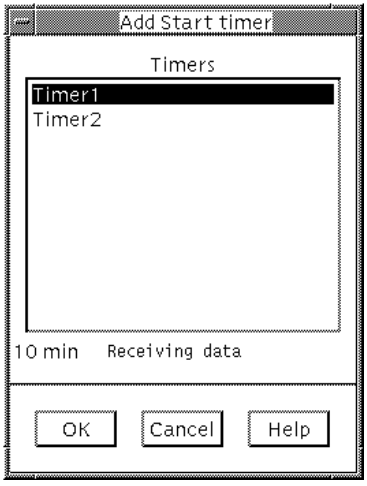


Figure 205: Add StartTimer Statement dialog

- When *OK* is clicked, a StartTimer statement is generated with the selected timer.

Data Dictionary > Add CancelTimer Statement

Generate a CancelTimer statement by selecting a timer.

A dialog which is analogous with *Add StartTimer Statement* will be displayed.

Data Dictionary > Add Attach Statement

Generate an Attach statement by selecting a test step and specifying an actual parameter list (if the test step has a formal parameter list).

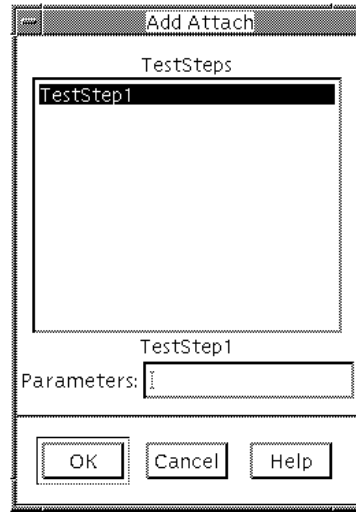


Figure 206: Add Attach Statement dialog

Data Dictionary > Add Timeout Statement

Generate a Timeout statement by selecting a timer.

A dialog which is analogous with *Add StartTimer Statement* will be displayed.

Reverting a Table

File > Revert

Causes all edits to the table **since it was last opened** to be discarded. In other words, the contents of the table are restored to what they were at the point when the table was opened.

This gives the TTCN Suite a **limited** undo capability. Note that the analyze status of the table may be set to *not analyzed* by this command.

Creating a New Constraint Table

The *Create Constraint* and *Copy Constraint* commands, facilitate the creation of Constraint tables:

Tools > Create Constraint

Creates a new Constraint table – related to the Type table currently shown in the Table Editor – in a new Table Editor window.

If you use `<Ctrl>` + *Create Constraint*, the new Constraint table will instead be shown in the current Table Editor.

Tools > Copy Constraint

Creates a new Constraint table – that is an exact copy of the Constraint table currently shown in the Table Editor – in a new Table Editor window.

If you use `<Ctrl>` + *Copy Constraint*, the new Constraint table will instead be shown in the current Table Editor.

Generating the Test Suite Overview Tables

The generation of Test Suite Overview Tables is described in "Generating the Test Suite Overview Tables" on page 1159 in chapter 24, *The TTCN Browser (on UNIX)*.

Using Popup Menus

The background popup menu is accessible when you click the right mouse button. When you click the right mouse button and <Ctrl>, the context sensitive menu will be available.

Context Sensitive Popup Menu

The context popup menu contains only one operation, *Find Table*, but in addition it also contains some of the content of the table to be opened by this Find Table operation. The identifier to use in the Find Table operation is retrieved from the field content under the mouse pointer when the context menu was invoked.

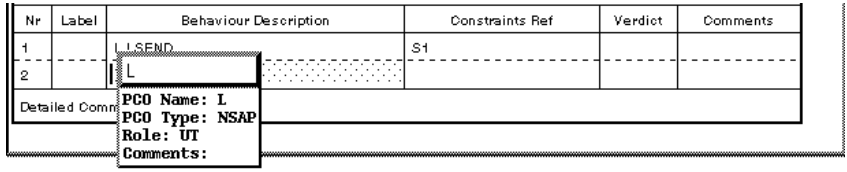


Figure 207: Example use of the TTCN Table Editor context popup menu

Background Popup Menu

The background popup menus are available in all the TTCN Suite tables but the appearance differs in accordance with the applicable operations. The popup menus in synchronized behaviour tables and desynchronized behaviour tables have a totally different appearance and will be described below:

Synchronized Behaviour Tables

If the Table Editor contains a behaviour table which is synchronized, the popup menu will display the same menu entries as the *SDT Link* menu. For an explanation of *synchronized* and the commands in the *SDT Link* menu see chapter 35, *TTCN Test Suite Generation*.

Desynchronized Behaviour Tables

If the Table Editor contains a behaviour table which is **not** synchronized, the popup menu will have a different appearance than in the previous case.

The popup menu will contain a sub menu containing all the menu entries of the *Data Dictionary* pull-down menu. It will also contain the following entries:

Cut Focused Row

Removes the focused row from the table and stores it in the clipboard.

Copy Focused Row

Copies the focused row to the paste buffer.

Paste Before Focused Row

Pastes the contents of the paste buffer **before** the focused row.

Delete Focused Row

Deletes the focused row.

Increase Indent

Increases the indentation of the **focused** behaviour line by one position.

Decrease Indent

Decreases the indentation of the **focused** behaviour line by one position.

Increase Sub Tree Indent

Increases the indentation of an entire **branch** in the behaviour tree by one position starting from the **focused** row.

Decrease Sub Tree Indent

Decreases the indentation of an entire **branch** in the behaviour tree by one position starting from the **focused** row.

Insert Row After

Insert a new row after the **focused** row.

Show Error Message

Displays the analysis status message (if any) of the **focused** field.

Find Table

Finds a named table.

Key and Button Bindings

Some bindings operate on the body rows as if they were organized as a tree (work only in the dynamic part of the TTCN document). The indentation level defines the tree structure.

Key	Action
`osfUp`	Move focus to a neighbor field.
`osfDown`	Move focus to a neighbor field.
`osfRight`	Move focus to a neighbor field.
`osfLeft`	Move focus to a neighbor field.
`Ctrl+osfUp`	Move focus to a neighbor field even when editing.
`Ctrl+osfDown`	Move focus to a neighbor field even when editing.
`Ctrl+osfRight`	Move focus to a neighbor field even when editing.
`Ctrl+osfLeft`	Move focus to a neighbor field even when editing.
`osfInsert`	Insert a new row, after focused row if already in body, otherwise last in body.
`osfHelp`	Get help on Table Editors.
`Ctrl+Z`	Run "Analyze & Stop" on page 1191 in chapter 26, *Analyzing TTCN Documents (on UNIX)*
`Ctrl+S`	Search forward with the current settings
`Ctrl+R`	Search backward with the current settings
`character keys`	Start editing field and then apply key.
`Delete`	Start editing field and then apply key.
`...`	Start editing field and then apply key.
`Tab`	Move focus to a neighbor field.
`Shift+Tab`	Same as `<Tab>` but in the opposite direction.

Key	Action
Ctrl+Tab	Navigate in the tool bar, the search bar, the scroll bars and the table
Ctrl+Shift+Tab	Same as <Ctrl+Tab> but in the opposite direction
Ctrl+space	Toggle selection of focused row
KP_Add	Increase indentation level of focused row
Ctrl+KP_Add	Increase indentation level of selected rows
Shift+KP_Add	Increase indentation level of focused subtree
Ctrl+Shift+KP_Add	Increase indentation level of subtree of single selected row
KP_Subtract	Decrease indentation level of focused row
Ctrl+KP_Subtract	Decrease indentation level of selected rows
Shift+KP_Subtract	Decrease indentation level of focused subtree
Ctrl+Shift+KP_Subtract	Decrease indentation level of subtree of single selected row
Ctrl+Shift+osfUp	Move focus to previous row at same indentation level
Ctrl+Shift+osfDown	Move focus to next row at same indentation level
Ctrl+Shift+osfRight	Move focus to last child row of focused row
Ctrl+Shift+osfLeft	Move focus to parent of focused row

Mouse button (OSF names)	Action
Button1	Set focus to the field, and start editing.
Button2	Move focus to the field.
Ctrl+Button1	Toggle selection status of the row
Ctrl+Shift+Button1	Toggle selection status of the subtree with the row as root, move the focus to the field under pointer

Chapter

26

Analyzing TTCN Documents (on UNIX)

This chapter contains a reference manual to the Analyzer in the TTCN Suite. A description of the *Find Table* command is also included. It can, for example, be used when you want to find erroneous tables that are displayed in the Analyzer log.

The TTCN Browser and the TTCN Table Editor are described in chapter 24, *The TTCN Browser (on UNIX)* and chapter 25, *The TTCN Table Editor (on UNIX)* respectively.

See chapter 3, *Introduction to the TTCN Suite (on UNIX), in the TTCN Suite 6.2 Getting Started* for an overview of the TTCN Suite tool-set.

> **Note: UNIX version**
>
> **This is the UNIX version of this chapter. The Windows version is chapter 31, *Analyzing TTCN Documents (in Windows)*.**

The TTCN Analyzer

The TTCN Analyzer in the TTCN Suite does a complete syntax check on both TTCN and ASN.1 (as it is used in TTCN). The Analyzer also performs a number of static semantic checks, mainly the uniqueness and existence of identifiers.

For implementation reasons, the Analyzer does not exactly follow the TTCN standard – some syntax restrictions and relaxations apply. These differences will not affect the vast majority of users, but for reference they are documented together with the standard syntax in chapter 38, *Languages Supported in the TTCN Suite*. This chapter also includes the static semantics supported by the TTCN Suite.

During analysis, the TTCN Suite compiles an extensive symbol table containing all named objects in the TTCN document and the references between them. This symbol table is used not only during the static semantic checks, but also by tools such as the Selector tool and the Compare tool.

The Analyzer can be called from various tools: from the Organizer, from a Browser, or from a Table Editor. It is possible to analyze an entire TTCN system, a whole TTCN document or just selected portions of a TTCN document (e.g. a single test case or all the PDU definitions).

The Analyzer considers a TTCN system to not contain any Flat View document, so it needs to be explicitly applied in order to analyze the flat view.

Using the Analyzer from the Organizer

Invoking the Analyzer from the Organizer is described in "Analyze TTCN" on page 118 in chapter 2, *The Organizer*.

Using the Analyzer from a Browser

The Analyzer works either on the entire TTCN system or on selected items in the Browser it is applied on. Selections can either be made by using the mouse or by using the Selector tool (see "Using More Complex Selections" on page 1124 in chapter 24, *The TTCN Browser (on UNIX)*.

Selections can be arbitrary – the TTCN Suite will analyze each selected item independently.

To analyze an entire TTCN document, do a *Select All* in the TTCN document browser and then apply the Analyzer. Alternatively apply the Analyzer on the TTCN document in the Organizer (see "Analyze TTCN" on page 118 in chapter 2, *The Organizer*).

Tools > Analyze

Conducts a syntax and static semantic check on the TTCN system or on the selected items (tables) in a Browser.

If an item is analyzed as incorrect, this will be recorded as an error message in the Analyzer log. Erroneous items are indicated by a black analysis bar in the Browser. On a color display this bar will be red.

Showing Progress

The Analyzer will, if invoked for a document or a whole system, display the progress made in the relevant Browser status window. This mechanism also provides the means to abort the analysis while it is running. Aborting is accomplished by clicking in the status window.

Tools > Analyze & Stop

Conducts a syntax and static semantic check on the TTCN system or on the selected items (tables) in a Browser but stops on the first found error and shows the table containing that error in a Table Editor.

- To execute this command via the quick-button, `<Ctrl>` must be pressed.

Using the Analyzer from a Table Editor

When used from the Table Editor the Analyzer will be applied to the contents of that particular table.

Tools > Analyze

Conducts a syntax and static semantic check on the table contents.

If a field in a table is analyzed as incorrect this will be recorded as an error message in the Analyzer log. Erroneous fields are indicated by a black status bar and the field is shaded grey (red on a color display).

Tools > Analyze & Stop

Conducts a syntax and static semantic check on the TTCN system but stops on the first found error and replaces the table in the Table Editor with the table containing the found error.

- To execute this command via the quick-button, `<Ctrl>` must be pressed.

Note:

When using the Table Editor, the *On Field* menu choice in the *Help* menu displays the TTCN BNF syntax applicable for the relevant field.

The *Analyzer* Dialog

The Analyzer uses the same dialog when started either from a Browser or a Table Editor:

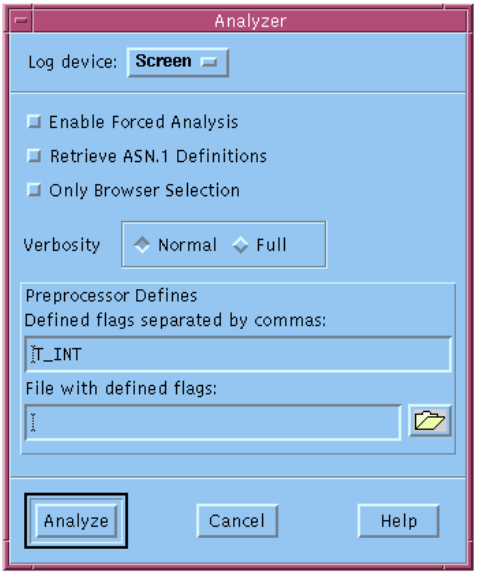


Figure 208: Analyzer dialog

Log Device

This option controls the log device for the Analyzer. The default value is *Screen*, but the log can be directed to a named file by choosing *File* or turned off altogether by choosing *None*.

For a full description of this command see "The TTCN Suite Logs" on page 6 in chapter 1, *User Interface and Basic Operations*.

Enable Forced Analysis

Indicates that all selected items are going to be analyzed and no item will be skipped. An item is normally skipped if it is OK.

Retrieve ASN.1 Definitions

Indicates that the definitions of encountered ASN.1 references should be retrieved. An ASN.1 definition is retrieved if this option is set or if the definition has not been fetched before.

Only Browser Selection

Check this item if the analysis shall be restricted to the Browser selection (this option is not applicable if started from a Table Editor).

Verbosity

Full verbosity gives more detailed information. The following extra information is available with full verbosity:

- Processing Cyclic references

  The items which will be left with incorrect references and incorrect analyze status are listed.

- Detect unused objects

  If applied on a complete test suite without finding any errors, an additional search for unused objects is performed.

Defined flags

The flags defined for the TTCN Suite preprocessor are separated by commas (see "TTCN Suite Preprocessor" on page 1205).

File with defined flags

This field can be used to point to a file with flags defined for the TTCN Suite preprocessor. Use comma (“,”) to separate the flags in this file.

Error Messages

Errors detected during analysis are recorded in the log. For the fields in the header of a table the error messages have the general format:

```
==================================
Table name and table type
Field type 1
Error message
Field type 2
Error message
:
:
Field type n
Error message
==================================
```

Example 205 ––

An error found in the Default Name (identifier) field of the Default Dynamic Behaviour called UT_DEFAULT:

```
==========================================
Analysis messages in table: UT_DEFAULT of type:
Default Dynamic Behaviour
 Default Name:
The referenced identifier UPPER_PCO is not declared.
UT_DEFAULT (p : UPPER_PCO)
----------------^
==========================================
```

––

In the body of a table (where the field type name does not uniquely define a field) the row number is also included:

- Row number
- Body type
- Error message

Example 206 ––

Errors found in a Test Step table header (the Default field) and in the body (rows 1 and 2 of the Behaviour Description):

```
==========================================
```

```
Analysis messages in table: ESTABLISH_CONNECTION of
type: Test Step Dynamic Behaviour
 Default:
Mismatched number of parameters: is 1, should be 0.
   Row: (#1)
      Behaviour Description:
The referenced identifier CR is of wrong type.
UPPER_PCO ! CR
------------^
   Row: (#2)
      Behaviour Description:
The referenced identifier CC is of wrong type.
UPPER_PCO ? CC
------------^
==========================================
```

Showing Error Messages

It is possible to display the Analyzer error message (if any) that is associated with the field in a table that has the current input focus. To do this, select *Error Messages* from the Table Editor *Show* menu.

Resolving Forward References

During analysis, the TTCN Suite does two main tasks:

- It checks the **syntax** of the abstract TTCN document.
- It checks some of the **static semantics** of the TTCN document.

The syntax of TTCN is defined in ISO/IEC 9646-3 Annex A as a list of BNF productions or rules. See also "The TTCN-MP Syntax Productions in BNF" on page 1574 in chapter 38, *Languages Supported in the TTCN Suite*. The TTCN Suite checks all these rules except for a very few minor deviations; for more information, see "The Restrictions in the TTCN Suite" on page 1570 in chapter 38, *Languages Supported in the TTCN Suite*). If a rule is not followed in the TTCN document then this will be reported as an error by the Analyzer.

The static semantic rules of TTCN are also defined in Annex A of ISO/IEC 9646. These are in the form of text statements which **limit** the use of some of the syntax rules, usually in a specific context. "TTCN Static Semantics" on page 1603 and "ASN.1 Static Semantics" on page 1619 in chapter 38, *Languages Supported in the TTCN Suite* describe the static semantics which are controlled by the Analyzer. The static semantics currently supported are mainly of the following types:

- Identifiers (names) are checked for uniqueness with respect to the scoping rules defined in ISO/IEC 9646-3
- The existence of referenced TTCN objects (e.g. a Test Step) from another TTCN object (e.g. Test Case) is verified
- Type control and type restriction control

In order to achieve this, the Analyzer constructs a symbol table of all the named TTCN objects that it finds during analysis, e.g. variable names, type identifiers, tables, etc.

This symbol table is used by some of the other tools, such as the Selector and the Reporter. This explains why it is necessary to ensure that the TTCN document is analyzed **before** applying these tools – the symbol table is updated each time the TTCN document is analyzed.

It is important to note that TTCN allows *forward reference*s. For example a Test Step may attach another Test Step that is declared **later** on in the TTCN document, that is, the referenced (i.e. attached) Test Step, appears **after** the Test Step that has done the attach.

The Analyzer has a feature called *back-pass* designed to resolve forward references. The back-pass is automatically invoked at certain points in the analysis process.

Forward references usually appear in the context of ASP/PDU definitions, constraints and as the result of attachment statements in behaviour trees. This is the reason that the Analyzer does not traverse the items to be analyzed in exactly the listed order. The Analyzer order deviates from the listed order in a few places:

- Items in the PDU Type Definitions sub-tree are analyzed before items in the ASP Type Definitions sub-tree.
- Items in the PDU Constraint Declarations sub-tree are analyzed before items in the ASP Constraint Declarations sub-tree.
- Items in the Defaults Library sub-tree are analyzed before items in the Test Step Library sub-tree and those are analyzed before items in the Test Cases sub-tree.
- Items in the Declarations Part, Constraints Part and Dynamic Part are analyzed before any items in the Import Part (where applicable) and overviews.

ASN.1 External Type/Value References

References to ASN.1 definitions in external ASN.1 modules can occur in the following four TTCN tables:

- Test Suite Constant Declarations By Reference
- ASN.1 Type Definitions By Reference
- ASN.1 ASP Type Definitions By Reference
- ASN.1 PDU Type Definitions By Reference

In these tables there are three important fields to consider:

- *Module Identifier* (set by user)
  The name of the ASN.1 Module. This should be the real name of the model, i.e. the name stated inside the module itself.
- *Type Reference/Value Reference* (set by user)
  This is the name of the desired type (or value).
- *Type Definition/Value* (set automatically when fetched)
  This is the field where the fetched type definition/value is copied into.

The term *definition* is used below both to denote an ASN.1 type definition and an ASN.1 value.

When an ASN.1 reference is analyzed, the referred definition must be available to the Analyzer. In the pro formas for this kind of reference in the TTCN Suite, there is an extra column where both the definition and the parse tree is stored.

Since the ASN.1 module is accessible from the Organizer the Analyzer can fetch the definition. If the Analyzer encounters any reference to a type/value, the definition is fetched, given the type/value name and the module identifier, and copied into the external field before the field is analyzed. The operation of fetching the definition of the reference is controlled by an option in the Analyzer dialog. Important to note is that the fields *Module Identifier* and *Type/Value Reference* are not parsed. This means that if a change is made in either of these two fields, the row will not be analyzed and thus no fetching will be made. The solution to this is to set the *Enable Forced Analysis* in the Analyzer dialog when altering a reference. See "The Analyzer Dialog" on page 1192.

To use external ASN.1 module references in the TTCN Suite, information about the referred ASN.1 module must be available. That is why the ASN.1 module must appear in the Organizer and a *dependency* must be defined from the ASN.1 Module to the TTCN document. See "Dependencies" on page 138 in chapter 2, *The Organizer*.

Analysis can be invoked either from the TTCN Suite or from the Organizer. The only difference, regarding dependencies, is that when the analysis is made from the TTCN Suite the dependencies are fetched explicitly while the Organizer always precedes an analysis with a dependency update.

Analyzing ASN.1 References

This is the general algorithm when encountering an ASN.1 in the analysis phase.

1. First of all, an attempted fetch is only made if no current definition exists or if the Analyzer fetching-option is selected.
2. The actual fetching is made, see "Retrieving External ASN.1 Definitions" on page 1198.
3. If the definition is successfully fetched, the identifiers in the definition is converted to TTCN compatible syntax, see "Syntactic Conversion" on page 1198.
4. Finally a possible update of the definition is made and the analysis is continued.

Retrieving External ASN.1 Definitions

From the ASN.1 module a simple parse tree is built to access its contents. The parse tree is (re-)built when either it has not been built before or the ASN.1 module has been modified since last access. The parse tree for an ASN.1 module "lives" for the duration of the whole analysis phase.

Syntactic Conversion

The received definition must be manipulated in the following sense. All identifiers must be syntactically checked and altered if they include any characters that is disallowed in TTCN. The rule is that all '-' (dash) characters in an ASN.1 name is replaced by a '_' (underscore).

No special care has to be considered to ASN.1 comments since they are ignored when the parse tree is built from the module.

External type references in the type definition (e.g "... Module.Type...") will only be converted like above and left for the Analyzer to deal with (see also "Restrictions" on page 1200).

Restrictions

These features are not supported in TTCN:

- External type reference identifiers on the form *ModuleIdentifier.Type/Valuereference* are not supported.
- Type/value references within the same ASN.1 module are not supported.

Furthermore there are restrictions in the ASN.1 Utilities when it is used to extract the external types and values. These restrictions are introduced in translating ASN.1 to SDL but since the same algorithm is used for extracting types and values, the restrictions are also relevant for the TTCN Suite. For more information, see "Translation of ASN.1 to SDL" on page 704 in chapter 13, *The ASN.1 Utilities*.

Error Handling

Any error that occurs while finding, parsing or accessing the ASN.1 module will cause the old definition to remain. Only when a completely successful fetch has been made, the definition will be updated. Possible errors could be:

- The ASN.1 Module not found among the dependencies.
- The ASN.1 Module file could not be found or was not readable.
- The ASN.1 Module file was not syntactically correct.
- The referred type/value could not be found in the module.

See chapter 13, *The ASN.1 Utilities* for a more general view on ASN.1 usage.

TTCN Static Type Restriction Control

The TTCN standard defines a set of semantic rules for type definitions. In particular it defines simple types as types which might contain a true subset of the values which the parent type might contain. The Analyzer now analyzes the TTCN type system and also reports violations of these rules. This analysis is implemented as a post-pass over the related tables, and is only applied when the previous passes have been successful.

Furthermore, restriction control is performed on a number of other constructs in the TTCN language, thus facilitating the task of writing semantically correct TTCN documents. Most of the TTCN types and values are checked and thereby reducing the chance of programming errors

slipping through to other tools which depends upon the correctness of the TTCN document.

Application of Static Type Restriction Control

Static Type Restriction Control is applied to at least the following TTCN tables and fields:

- Simple type definitions
- TTCN structured type / ASP / PDU / CM definitions
- Test Suite Constant Declarations
- Test Suite Variable Declarations
- TTCN Structured Type / ASP / PDU / CM Constraint declarations
- Actual constraint reference parameter lists
- Actual parameter lists

The Static Type Restriction Control also evaluates expressions in advance, where possible, and checks that the result does not violate the type restrictions of the field where the value is found. It operates on both types and values, regarding specific values as special cases of types.

Reducing Level of Restriction Control

Setting the environment variable `ITEX_RESTRICT_LEVEL` to the value `NONE` will disable the strict checking of type restrictions. This might be useful for allowing test suites with deliberate type errors pass through the analysis. A setting of `SOME` will relax the analysis somewhat, and is recommended before trying `NONE`. In order to get the most strict analysis (default), the value `STRICT` might be used.

Generating a Flat View

A Flat View is the part of the modular TTCN Suite environment that presents the tree structure of an expanded modularized TTCN source. This complete view includes all explicit defined objects in the TTCN system.

The Flat View interactive interfaces is similar to the Browser. Unlike the Browser, the Flat View is started (generated) from the Organizer.

The structure of the TTCN source in a Flat View is similar to a Test Suite. Neither the Import tables nor Export tables are displayed in a Flat View.

Generating a Flat View in a system, requires that at least the root document of the system to be connected to a file. The connected document must be of one of these types:

- Test Suite
- Modular Test Suite
- TTCN Module

If this condition is not fulfilled, the operation Generate Flat View fails. Also if the current Flat View is modified or is locked (by any user) the operation fails.

If a Flat View is edited by the user, it must be saved in another document file. The user is asked for a new file name. Otherwise, if the existing Flat View file is not modified, it is overwritten by the new generated Flat View if the generation is redone.

For each TTCN system in the organizer only one instance of Flat View can be generated. A Flat View can be regenerated. A Flat View includes all explicit defined objects in the system at the time of the (re)generation.

The first time the Flat View in a system is generated, the name of the Flat View file is the same as the file name of the root document in the system if the name is not already used. The file extension for a Flat View file is `.ifv`.

An already existing Flat View is always connected to a document file. If this existing Flat View is in the current target directory (and neither is modified nor locked) the Flat View file is reused (the Flat View is regenerated). Otherwise a new Flat View file in the current target directory is used.

Algorithm

When the command Generate Flat View is invoked the following cases may occur:

No Flat View is Available

In other words this is the first time a Flat View is generated for the given system. A new Flat View is generated in the target directory. The name of the document file is generated. This name is derived from the file name of the root document in the system. A Flat View icon is added to the selected TTCN system in the Organizer and it is connected to the generated document file.

A Flat View is Available

If the file connected to the existing Flat View either is locked or modified the operation fails. If the existing file is not in the target directory do as the previous case except not to add a Flat View system node.

Otherwise if the existing file is in the current target directory the file is reused (the current document file name must have been derived from the file name of the root document in the system).

If the Flat View is modified, the operation (Generate Flat View) fails and the user is asked to save the Flat View (into another file) before regenerating.

Save a Flat View

A user modified Flat View must be saved as a document of type (non-modular) Test Suite in another file than the Flat View file.

When the user invokes the save operation from the Organizer, a new file name must be given otherwise the save operation fails. If the user invokes the save operation from the TTCN Suite a file browser pops up and the user is asked for a file name.

Merging Objects into a Flat View

A Flat View contains a copy of all explicitly defined objects in a system. When a Flat View is generated for a TTCN system, explicitly defined objects in each system node is merged into the Flat View. The merged objects may have the same name.

Objects of the following types are treated special:

Overview Tables

The main table in the Overview Part is the Test Suite Structure table. Depending on the type of the root node in the system the Overview tables in the generated Flat View contain the following information.

Test Suite (Modular or Non-Modular)

This type of system nodes have a Test Suite Structure table. The header of the corresponding table in the Flat View inherits the information in the header of the root table.

The Detailed Comments in the index tables are copied to the index tables in the Overview part of the generated Flat View.

TTCN Module

In this case the TTCN Module Exports table is the corresponding table to the Test Suite Structure table in the Flat View. The TTCN Module Exports contains two extra header fields: Objective and TTCN ModuleRef. These two fields are ignored when the Flat View is generated.

The Detailed Comments in the index tables are copied to the index tables in the Overview part of the generated Flat View.

Package

In this case there is no corresponding table to the Test Suite Structure table in the generated Flat View.

Import and Export Tables

These tables contain no explicit defined object and therefore they are ignored.

External Tables

These tables contain no explicit defined object and therefore they are ignored.

Multiple Tables

The objects in the multiple tables are added to the corresponding multiple table in the generated Flat View. The Detailed Comment in the multiple table is added to the Detailed Comment in the corresponding multiple table.

Single Tables

Add each single table to the appropriate place in the generated Flat View.

Group Nodes

For each group node if it does not exist already in the generated Flat View create a new group node.

TTCN Suite Preprocessor

The TTCN Suite preprocessor is a text processor that manipulates the text of a TTCN table as part of the first phase of Analyzing. TTCN Suite preprocessor allows you to produce different ETS code depending on the flags set. The syntax of TTCN Suite preprocessor directives is similar to directives for a C preprocessor:

```
#if <expr>
....
[#else]
...
#endif

<expr> ::= defined(<flag>) | '!' <expr> | <expr>
'||' <expr> | <expr> '&&' <expr> | (expr)
```

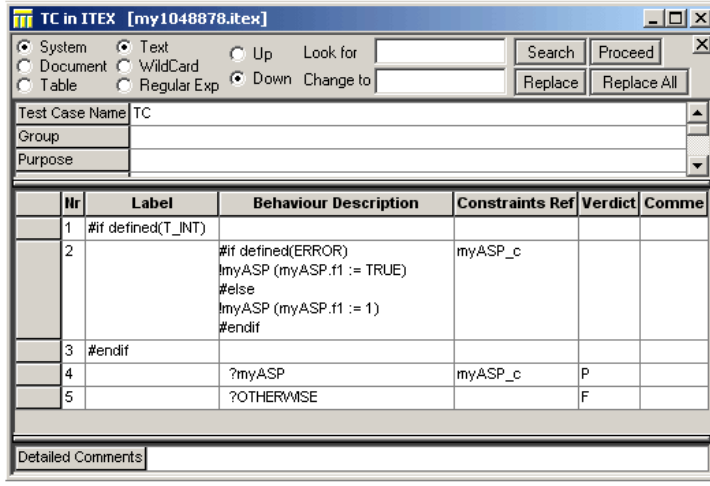


Figure 209: Example of preprocessor directives

These directives can be used in every editable field, not different from TTCN and ASN.1 definitions. The '`#if`' directive and its corresponding '`#endif`' directive must be placed within the same field, except when they are used in the body of a TTCN table that may consist of several rows. In this case it is allowed to use directives '`#if`', '`#else`' and '`#endif`' in separate rows, and only one such directive in the first field is allowed in such a row. The scope of such a directive will be all rows between.

The defined flags for Analysis and Code Generation are set in the Analyzer options (see "The Analyzer Dialog" on page 1192) The flags can also be set via options '-i' and '-I' from a command line session (see "Running the TTCN to C Compiler from the Command Line" on page 1323 in chapter 32, *The TTCN to C Compiler (in Windows)*).

Finding Tables

The Find Table tool searches for and displays named tables. The name searched for is given by the currently marked piece of text in any TTCN Suite window. This tool is very useful when debugging a TTCN document from an Analyzer log.

Finding Tables by Name

The Find Table tool is available from the *Tools* menu in the *TTCN* sub menu in the Organizer, from the *Tools* menu in a Browser and from the *Tools* menu in a Table Editor. Unlike most other tools it does **not** require a selection: it applies to the context given by where the tool is started (i.e. a TTCN table or a TTCN document), and its action is the same whether it is invoked from the Organizer, from a Browser or from a Table Editor.

Tools > Find Table

Finds a named table or the table in which a named object, such as a test case variable, is declared. The name searched for is given by the currently marked piece of text in a TTCN Suite window. If no text is marked Find Table will use the contents of the text clipboard (used for *Cut* and *Copy*).

A typical use of this tool would be to debug a TTCN document when analyzing. Erroneous tables identified in the Analyzer log window can be easily found by marking the names of these tables in the log window and applying the Find Table tool.

The Find Table tool is able to find named objects with the name specified as `<document name>::<object name>` (e.g. `Test_Suite_A::SEND`), where `<document name>` is the name of a document in the TTCN system and `<object name>` is the name of an object available in the context of that document. This may also be specified as `<document name>__<object name>` (i.e. with underscores instead of colons).

> **Note:**
>
> The Find Table tool will in fact use marked text as the look-up string from any X Window, not just TTCN Suite windows.

Example 207 ––

Marking the PDU name *CON_req* in, say, the behaviour line *L! CON_req (X:=1)* of a test step, and then choosing the Find Table tool will cause the PDU definition of CON_req to be displayed (assuming it exists).

On the other hand, marking the test case variable *X* in the same line would cause the Test Case Variables table, in which *X* is declared, to be displayed.

––

Chapter

27

The TTCN to C Compiler (on UNIX)

This chapter describes what the TTCN to C compiler is used for, how to run it and the structure of the generated code.

When the TTCN to C compiler has translated TTCN into C code, the code must be adapted with the system that is to be tested. The adaption process is described in chapter 36, *Adaptation of Generated Code*.

Note: ASN.1 support

The TTCN to C compiler supports only a limited subset of ASN.1. See "Support for External ASN.1 in the TTCN Suite" on page 733 in chapter 13, *The ASN.1 Utilities, in the User´s Manual* for further details on the restrictions that apply.

Note: UNIX version

This is the UNIX version of the chapter. The Windows version is chapter 32, *The TTCN to C Compiler (in Windows)*.

Introduction to the TTCN to C Compiler

When developing new systems or implementations of any kind, the developing process is divided into well defined phases. Most commonly, the first phases involve some kind of specification and abstract design of the new system. After a while, the implementation phase is entered and finally, when all parts are joined together, the test phase is activated.

In any case when a system is tested, we want to make sure that its behavior conforms to a set of well defined rules. TTCN was developed for the specification of test sequences. Unfortunately, as very few systems interpret or compile pure TTCN, we need to translate the TTCN notation into a language which can be compiled and executed. In the case of the TTCN Suite, the TTCN to C compiler translates TTCN to ANSI-C.

Even after the TTCN code has been translated, there are a couple of things that need to be taken care of. In this case, we must *adapt* the generated code with the system it intends to test. This chapter describes how the TTCN to C compiler is used. The adaption process is described in chapter 36, *Adaptation of Generated Code*.

Getting Started

Unfortunately a test sequence description expressed in TTCN can not easily be executed as it is. This is because the test notation is not executable and only few test environments interpret pure TTCN. A different approach to create an executable test suite (ETS), is to translate the formal test description into a language which can be compiled into an executable format.

The TTCN to C compiler translates TTCN into ANSI-C which can be compiled by an ANSI-C compiler. Figure 210 depicts the first step in the process of creating an ETS using the TTCN to C compiler.

The generated code, called the TTCN runtime behavior, is only one of the two major modules of an ETS.

The second module which is needed includes test support functions which are dependent on the protocol used, the host machine, test equipment, etc. For this reason, it is up to the user to write this second module and in such way adapt the TTCN runtime behavior to the system he/she wants to test.

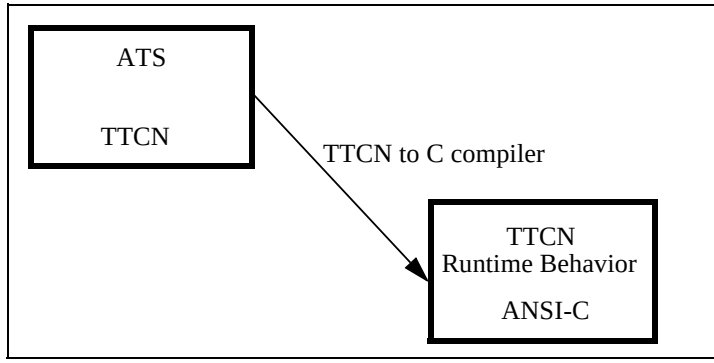


Figure 210: Translating TTCN to ANSI-C

The adaption process is described in chapter 36, *Adaptation of Generated Code*. Figure 211 displays the anatomy of the final result.

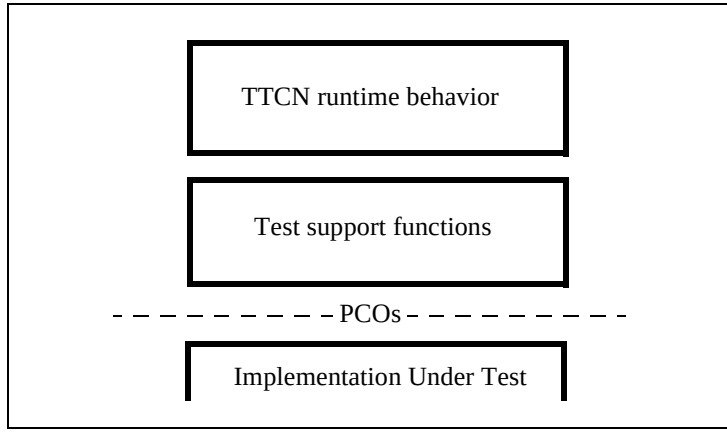


Figure 211: The anatomy of an ETS

Running the TTCN to C Compiler

The TTCN to C compiler can be activated either from the TTCN Browser, both as a quick button and a menu command, or from the Organizer when a TTCN test suite is selected.

> **Note:**
>
> Observe that the TTCN to C compiler is unable to function correctly if the test suite is not fully verified correct. This verification is accomplished by running the Analyzer tool on all the parts of the test suite.

Tools > Make

When you select *Make*, the TTCN Suite Make dialog is opened. The dialog contains two important toggle buttons, *Analyze & Generate* and *Compile & Link*.

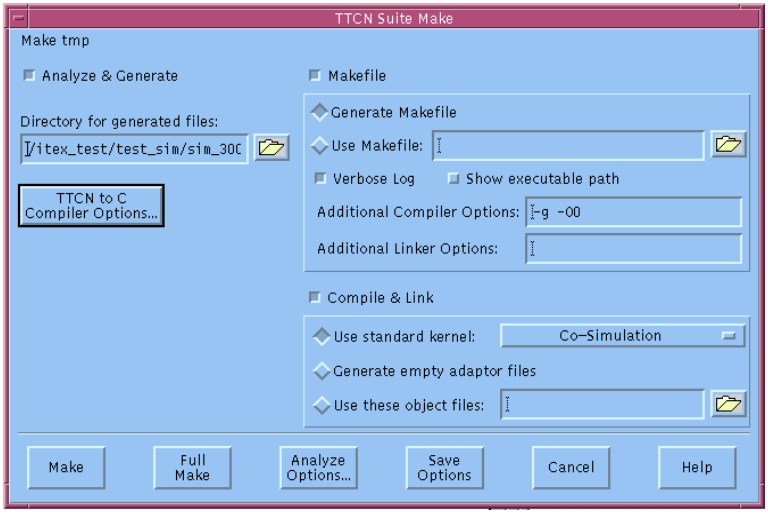


Figure 212: Dialog for starting the TTCN to C compiler

Analyze & Generate

If *Analyze & Generate* is set, the Analyzer will first be called to analyze the test suite before any code is generated. If errors are found, these will be reported and no code will be written to file. The generation phase

performed after the analysis phase will only generate code for the parts that need to be re-generated.

Directory for generated files

If code is generated, that code will reside in the directory specified in this field.

Makefile

If *Compile & Link* is set, the TTCN Suite will also take care of the making and linking of the generated code. At this point you have the possibility to specify how you want the code to be built. Either you can specify that the default generated makefile is to be used, or your own special makefile. This is done by setting the radio buttons in the field below the *Makefile* toggle button.

Verbose Log

The *Verbose Log* switches the logging between the default mode, which prints the result of the Make process as it goes along, and the *Verbose* mode, where all possible information is presented. The *Verbose* log is primarily intended for debugging.

The *Verbose* log contains the following information in addition to the default log:

- signal parameters, with an attempt to restore all names (for signal, fields, parameters, choices, etc.), even when these names are lost (which is the case when the user construct the signals directly in the adaptor). The default log can be made to show this if the Compilation flag /DSHOWSIGNALPARAMETERS is set.
- matching processes are presented *Verbose*. Each field shows how it was matched, using the indentation level for nested fields.

Show Executable Path

When *Show Executable Path* is selected, the lines in the ETS log will be prefixed by the path for the executing Test Case and Test Steps, like the following example:

```
Test Case:[line number]  Step_no:[line number] ... Test Step_n
```

Compilation flag /DSHOWSIGNALPARAMETERS

The flag `SHOWSIGNALPARAMETERS` is generated in the Makefile even when the log is not *Verbose*. With this compilation flag the default log will also contain the complete contents of all signals, just like when log is set to *Verbose*. When this is not desired, this flag can easily be removed from the Makefile.

Additional Compiler Options and Additional Linker Options

Here it is possible to specify any compiler and linker options that will be added into the generated Makefile.

Compile & Link

The compiler needs to know what files to link into the executable. Three different possibilities are at hand:

- You have not yet written an adaptor, but you want an empty template to allow compilation to go through successfully.
- You have already written an adaptor which you want to link into the executable.
- You want to link to a simulator kernel to allow you to simulate.

The TTCN to C compiler assumes the first case to be the default one. In this case the *Generate empty adaptor files* radio button is set. If you do not have adaptor files (`adaptor.c` and `adaptor.h`) in the target directory, these will be created, as empty templates, for you. If you already have generated and edited adaptor files these will of course not be overwritten.

If you already have your own compiled adaptor files elsewhere in the file system, these files, as object files, should be listed in the text field below the *Use the object files* dialog button. Observe that you should use absolute path names for the adaptor files to avoid problems for the makefile to find these files! An absolute path name is a path name that identifies a file uniquely from the root of the files system, for example `/home/myhome/myadaptor.o`.

Finally, if you want to generate code for simulation, the *Use standard kernel* radio button should be set with the option *Simulation*. This is the only available kernel at the moment so no other choice is valid here.

Make

If the *Make* button is pressed, the compiler will only generate code for the parts that really need to be re-generated.

Full Make

If the *Full Make* button is pressed, everything will be regenerated again.

When the generation phase is started, a new log window will be displayed where information about the compiler actions will be displayed.

TTCN to C Compiler Options

By pressing the *TTCN to C Compiler Options* button in the Make dialog, the dialog depicted in Figure 213 will be displayed.

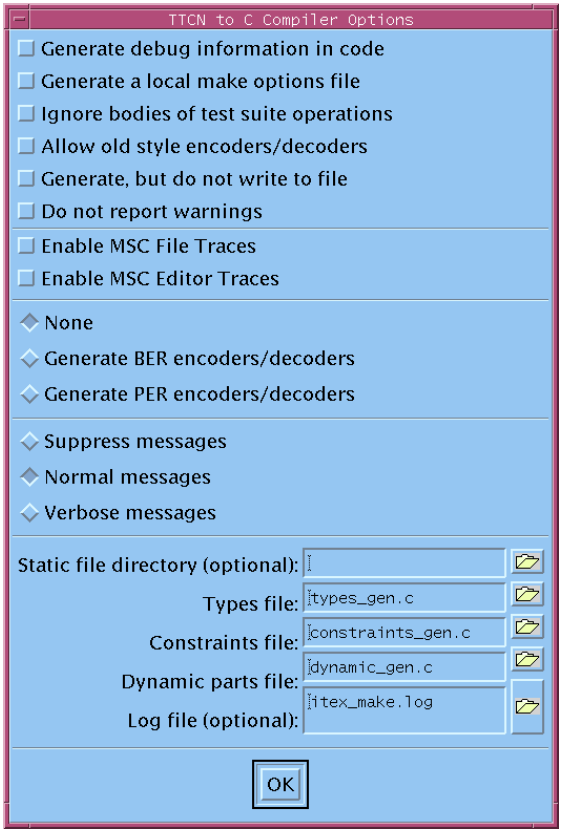


Figure 213: Setting extra TTCN to C compiler options

- If the *Generate debug information in code* button is set, the compiler will generate extra code for tracing and debugging.
- If the *Generate a local make options file* button is set, the compiler will generate a file called `makeopts` in the target directory. This file is used to define settings used by the makefile. You have the flexi-

bility to choose compiler and add extra information for the make process.

- If the *Ignore bodies of test suite operations* button is set, the compiler will ignore to generate code for test suite operations. This is useful if you already have a file with your own test suite operations that only need to be linked to the rest of the code. If the button is not set, the test suite operators will be generated and included in the file `tsop_gen.c`.
- If the *Allow old style encoders/decoders* button is set, the compiler will generate two extra files that will allow you to use an adaptor written for the old code generator to be used with the new one.
- If the *Generate, but do not write to file* button is on, the compiler will not generate code. This is useful for users that only wish to generate code when they know no generation errors will occur.
- If the *Do not report warnings* button is set, the compiler will not inform you about warnings.

MSC Trace

- *Enable MSC/PR file tracing*

  Generates MSC/PR format files that are saved in the current working directory. By default, the MSC file names will be on the form `log_<TestCaseId>_<SequenceNo>.mpr`, where `<TestCaseId>` is substituted by the test case name of the logged test case. `<SequenceNo>` is an integer that is started by 0000 and increased by one if there is already a version n in the working directory.

  The preprocessor constant MSC_FILE_MODE should be defined at compile-time of `mscgen.c` to get this mode. The constant GENMSC should be defined for compiling `globalvar.c` to activate appropriate calls.

  If the file cannot be created, a new attempt will be done at the start of the next test case. No file log will be created.

  The function `MscSetPrefix` can be used to change the path and prefix of generated files at runtime.

- *Enable MSC Editor tracing*

Creates a new diagram in the MSC Editor when a new test case is started and then appends and displays the events as they are executed. This mode assumes that an MSC Editor license is available and that TTCN Suite is running at the host where the ETS is running and provides run-time MSC logging.

The preprocessor constant MSC_MSCE_MODE should be defined at compile-time of `genmsc.c` to get this mode. The constant GENMSC should be defined for compiling `globalvar.c` for activation of the appropriate calls.

If the creation of events in the MSC Editor fails, it will be retried at the next event, possibly creating an inconsistent MSC.

This mode may also require access to the Public Interface libraries and include files that can be found in the installation.

For more information, see "TTCN Test Logs in MSC Format" on page 1328.

Encoders

- By selecting one of the radio buttons *None*, *Generate BER encoders/decoders* or *Generate PER encoders/decoders* the compiler will either omit generation of encoders/decoders or generate extra code for BER or PER encoding and decoding support. See chapter 58, *ASN.1 Encoding and De-coding in the SDL Suite, in the User´s Manual*.

Encoders

- By selecting one of the radio buttons *Suppress messages*, *Normal Messages* or *Verbose messages* you can set the level of verbosity when the compiler generates code. The default value is *Normal messages*.

Files

You can also choose the names of the files you want to use. This is done in the fields in the lower part of the TTCN to C compiler options dialog.

- The optional *Static file directory* field should only be used by users that brutally want to change the behavior of the compiler by using their own static files. For "normal" users it is recommended that this field is not set.

- The *Types file* field determines the name of the file to which type definitions from your test suite are generated. The default value is `types_gen.c`.
- The *Constraints file* field determines the name of the file to which constraint definitions from your test suite are generated. The default value is `constraints_gen.c`.
- The *Dynamic part file* field determines the name of the file to which the dynamic part from your test suite is generated. The default value is `dynamic_gen.c`.
- The *Log file* field determines the name of the file to which messages about the generation are to be written. Default is standard out.

Running the TTCN to C Compiler from the Command Line

Please see "Running the TTCN to C Compiler from the Command Line" on page 1323 in chapter 32, *The TTCN to C Compiler (in Windows)*.

What Is Generated?

In the case of the code generated from the compiler we also need a set of static functions which handle TTCN basics and other internal events. Even if these functions are vital for the successful compilation and execution of the generated code, the user should not have to worry about this part. These functions are gathered in a small set of static files which are compiled by the generated makefile and linked with the rest of the code.

The Code Files

The generated makefile is the file containing a definition of how the code should be compiled and linked.

The `adaptor.h` and `adaptor.c` files are the files that contain the adaptation code. If code is generated for the first time, these files will be generated by the compiler with empty function templates for the user to implement. On the other hand, if these files are present in the target directory the user does not have to worry about getting them overwritten.

The `*_gen.{c,h}` files contains the code generated for the TTCN test suite.

The `asn1ende.h` file contains the encode and decode functions for the ASN.1 Types. See chapter 58, *ASN.1 Encoding and De-coding in the SDL Suite, in the User´s Manual*. (Only generated if ASN.1 encoding/decoding support has been selected)

The Adaptation

We are now ready to deal in greater detail with the adaptation phase which is the final phase to create an executable test suite. The adaptation process is described in "Adaptation of Generated Code" on page 1487 in chapter 36, *Adaptation of Generated Code*.

TTCN Test Logs in MSC Format

Please refer to "TTCN Test Logs in MSC Format" on page 1328 in chapter 32, *The TTCN to C Compiler (in Windows)*.

Chapter

28

The SDL and TTCN Integrated Simulator (U)

This chapter describes theSDL and TTCN Integrated Simulator in the UNIX version of the TTCN Suite. It gives an introduction to the concept of the SDL and TTCN Integrated Simulator, as well as a guide to its functionality and an introduction to using the SDL and TTCN Integrated Simulator for controlling real Executable Test Suites.

> **Note: UNIX version**
>
> **This is the UNIX version of the chapter. The Windows version is chapter 33, *The SDL and TTCN Integrated Simulator (W)*.**

The SDL and TTCN Integrated Simulator

The SDL and TTCN Integrated Simulator allows you to:

- Execute a compiled test suite and inspect the results.
- Execute test cases and test groups by stepping through the TTCN lines or executing at full speed with the possibility to set breakpoints at given tables/lines.
- View the actions performed by the various parallel test components (PTCs) in multiple windows if needed when executing a concurrent test suite.

The possibilities to pinpoint the cause of the test result beyond the test verdict *fail* are vast, and will undoubtedly help in improving the time needed for testing in the development phase.

Performing a Integrated-Simulation

Given a test suite containing a set of tests you want to execute together with a simulated SDL system, the following steps are to be performed:

1. Analyze the test suite and make sure it contains no errors. (This can be done using the TTCN Suite *Make* dialog.)
2. Using the TTCN Suite *Make* dialog, build an executable test suite with the *Use standard kernel* radio button set, and with the *Integrated-Simulation* kernel selected.

3. Start the SDL and TTCN Integrated Simulator by pressing the integrated-simulation quick-button on the top of the TTCN Browser, or selecting *Start Integrated-Simulator* from the Browser *Tools* menu.

Setting Up an SDL and TTCN Integrated Simulation

A complete test of a system design can be performed prior to building a prototype by designing the system in SDL, and using the TTCN Suite to write test specifications in TTCN, or using a tool to automatically or semi-automatically generate test suites from a system description in any language. The actual testing is then a matter of connecting the two simulators.

To achieve the connection, follow these steps:

1. Generate and start the SDL Simulator.
2. Start the SDL and TTCN Integrated Simulator by selecting the Integrated-Simulator quick-button from the tool bar of the Browser.
3. When the SDL and TTCN Integrated Simulator has appeared, along with a selection of executable test cases and test groups, enter the command
   `start-itex` in the SDL Simulator.
4. Select one or more test cases or test groups in the SDL and TTCN Integrated Simulator Setup window, and click the *Run Test* button.
5. In the SDL Simulator, give any of the start commands (**`Forever`**, **`Go`**...).

The loaded SDL system is now being simulated on the SDL Suite side, and conformance tested from the TTCN Suite side with the SDL and TTCN Integrated Simulator.

The SDL and TTCN Integrated Simulator User Interface

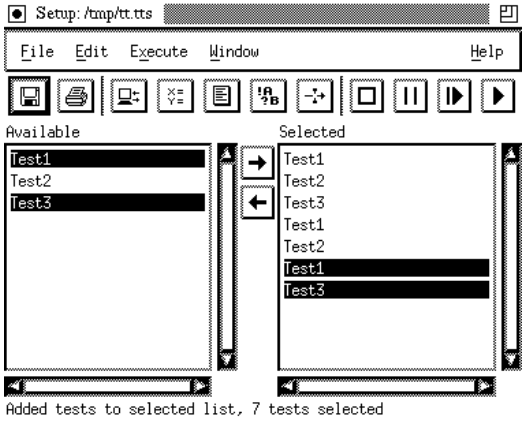


Figure 214: The SDL and TTCN Integrated Simulator Test Setup window

The Test Setup window is the main window of the SDL and TTCN Integrated Simulator. It manages an Executable Test Suite and a number of associated documents. The main part of the window is used for test selection. The left list depicts the currently available tests, and the right list depicts a selection of test cases and/or test groups.

Managing Setup Documents

The menu choices in the *File* menu are used for creating and saving documents. For information on the operations not mentioned here, see the general description of the *File* menu in "File Menu" on page 8 in chapter 1, *User Interface and Basic Operations*.

File > New

Creates a new Setup document. The new setup inherits the Executable Test Suite from the current Setup.

File > Open

Opens a previously saved Setup file in a new window.

File > Save

Saves the current Setup file. If the Setup was not previously saved, a name will be prompted for. A saved Setup consists of the file names of the associated documents, as well as the ETS name.

File > Save As...

Saves the Setup file with another file name. A running ETS might need to be restarted in order for changes to become effective.

File > Restart ETS

Restarts the Executable Test Suite / SDL and TTCN Integrated Simulator. This may compromise the consistency of running test cases, but might be necessary in order to make changes to the Setup effective.

File > Close

Closes the current view of the Setup document. If there are no more views of the Setup, the ETS will be terminated along with any test cases. If the Setup has been changed, a dialog will prompt for a file name for saving the Setup before it is closed. If the Setup is the only one currently open, the program will quit.

Selecting Test Cases and Groups

This section lists the SDL and TTCN Integrated Simulator commands available in the *Edit* menu.

Edit > Documents > Select ETS

Enables you to select another ETS for the test Setup. This might be useful for selecting a new ETS, using the same Setup as the old one.

Edit > Documents > Add Document

Enables you to add a document type to this Setup. A standard set of document types are provided by default at program start-up.

Edit > Show > Test Cases

Sends a command to the ETS such that it lists all Test Cases in the *Available* list.

Edit > Show > Test Groups

Sends a command to the ETS such that it lists the Test Groups.

Edit > Add All

Adds all tests from the set of available tests to the set of selected tests.

Edit > Add

Adds the tests currently selected in the *Available* list, to those present in the *Selected* list. Both test cases and test groups may be present in the *Selected* list at the same time.

Edit > Remove

Removes the selected tests from the *Selected* list.

Edit > Remove All

Removes all tests from the *Selected* list.

Executing a Test

This section lists the SDL and TTCN Integrated Simulator commands available in the *Execute* menu.

Execute > Test > Cancel

Cancels a test in progress. All information will be lost and the ETS returns to its initial state. This might compromise the consistency of a test since it might leave the IUT in any state.

Execute > Test > Pause

Temporarily pauses the test execution. This might be used to examine the state of the IUT or of the ETS.

Execute > Test > Step

Steps one line of TTCN code. In a simulated environment this is a well-defined operation, as opposed to a real test system. In the latter case, timing might be compromised.

Execute > Test > Run test

Runs/Continues a test at full speed until it is finished or until a breakpoint is reached.

Execute > Breakpoints > Breakpoints

Shows the breakpoint editor, as described in "The SDL and TTCN Integrated Simulator Editor" on page 1230. Notice: If the quick button is disabled, this is because no breakpoint document is currently associated with the Setup. Use this menu item to add such a breakpoint document to the Setup.

Execute > Breakpoints > Enable

Enables all breakpoints (default).

Execute > Breakpoints > Disable

Temporarily disables all breakpoints.

Execute > Monitor > PCOs

Monitors all PCO Queues in a separate monitor window.

Execute > Monitor > Timers

Monitors all Timers in a separate monitor window.

Execute > Monitor > PTCs

Monitors all CPs in a separate monitor window.

Execute > Monitor > CPs

Monitors all CPs in a separate monitor window.

Execute > Enable ITEX Tracking

Highlights the next statement line to be executed in the Table Editor. This is on condition that you are running an SDL and TTCN Integrated Simulator session and that the SDL and TTCN Integrated Simulator was started from within the TTCN Suite.

Viewing Documents

This section lists the SDL and TTCN Integrated Simulator commands available in the *Window* menu.

Window > New Window

Enables you to display a new view of the same Setup, with a possibly different test selection. No new Setup is created with this command.

Window > Execution Trace

Shows a new view of the Test Execution trace. If the item is not accessible, select *Document* and then *Add Trace Document* in the *Edit* menu to add a document of this type to the current Setup.

Window > Conformance Log

Shows a new view of the Test Conformance Log. If the item is not accessible, select *Document* and then *Add Log Document* in the *Edit* menu to add a document of this type to the current Setup.

Window > Configuration Editor

Shows a new editor for the Test Configuration, see "The SDL and TTCN Integrated Simulator Editor" on page 1230. If the item is not accessible, select *Document* and then *Add Configuration Document* in the *Edit* menu to add a document of this type to the current Setup. The Test configuration is system dependent and not used for integrated-simulation.

Window > Parameter Editor

Shows a new editor for the Test Suite Parameters, see "The SDL and TTCN Integrated Simulator Editor" on page 1230. If the item is not accessible, select *Document* and then *Add Parameter Document* in the *Edit* menu to add a document of this type to the current Setup.

The SDL and TTCN Integrated Simulator Editor

A generic editor is present for editing some of the documents, as for instance the Breakpoints, Parameter and the Configuration. It is of limited interest to those who are only interested in SDL and TTCN Integrated Simulation, but it might be of more interest if you consider using the GUI for running real executable test suites.

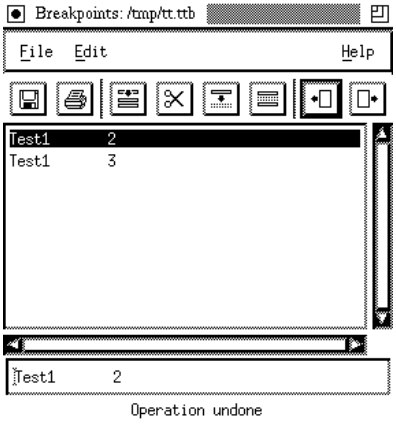


Figure 215: The SDL and TTCN Integrated Simulator editor for breakpoints

The editor is row oriented and the format of each row is determined by the current ETS. It is up to the ETS to warn the user for syntactic and/or semantic errors in the edited document. Please refer to the documentation of the ETS for more throughout definitions of the expected contents of each row.

Managing Documents in the SDL and TTCN Integrated Simulator Editor

The menu choices in the *File* menu are used for creating and saving documents. For more information, see the general description of the *File* menu in "File Menu" on page 8 in chapter 1, *User Interface and Basic Operations*.

Editing Documents in the SDL and TTCN Integrated Simulator Editor

The following section lists the SDL and TTCN Integrated Simulator Editor commands available in the *Edit* menu.

Edit > Undo

Undoes the last change of the document. This functionality is disabled when the document is saved or if a new document is opened. There are more than 200 undo steps.

Edit > Redo

Redoes the next undone change to the document. Applying a new operation removes any non-redone steps.

Edit > Cut

Cuts the currently selected row from the document. The row is placed in an internal clipboard.

Edit > Copy

Copies the currently selected row from the document to the internal clipboard.

Edit > Paste

Pastes the contents of the internal clipboard before the currently selected row.

Edit > Add

Inserts an empty row before the currently selected row.

Integrated-Simulation from command line

The generated ETS for Integrated-Simulation may be started from command line. Do this without any arguments in command line. In this case ETS has no any graphic interface. It reads commands from stdin and print the results of performed commands, Conformance Log, etc. to stdout. Extra debug output will be printed if the environment variable TTCN_SIMULATOR_DEBUG is set. The available commands are as follows:

breakpoint

`breakpoint <table name> <line number>`

Set the breakpoint in table `<table name>` on `<line number>`.

Example 208 ----------

```
breakpoint Init_step 1
```

---

breakpoints

`breakpoints`

List set of breakpoints.

cancel

`cancel`

Stop execution of test case.

delete

`delete [<table name> [<line number>]]`

Delete the breakpoint from table <table name> on the line <line number>. When <line number> is not set, all breakpoints from table <table name> will be removed. If <table name> also isn't set, all breakpoints will be removed.

Example 209 ----------

```
delete Init_step 1
```

---

gett

```
gett <value>
```

Print the value.

Example 210 ----------

gett VarX

---

glist

```
glist
```

List the available Test Groups.

list

```
list
```

List the available Test Cases.

pcos

```
pcos
```

List the available PCOs.

quit

```
quit
```

Exit the ETS.

run

```
run [<Test Case or Test Group>]
```

Start execution of the Test Case or Test Group. When Test Case is running and paused, may be used without arguments to continue Test Case execution.

Example 211 ----------

```
run TC
```

---

run-SDL (Windows only)

```
run-SDL <SDL Simulator executeable>
```

Start the SDL Simulator and redirect its input in order to pass commands to SDL Simulator. Wait until the SDL Simulator start and con-

nect to the Postmaster. When the TTCN Simulator exits, it automatically terminates any such started SDL Simulator.

Example 212 ----------

```
run-SDL cosim_smc.exe
```

SDL-command (Windows only)

`SDL-command <command>`

Passes the <command> to SDL Simulator, that previously has been started using the command `run-SDL`.

Example 213 ----------

```
SDL-command start-itex
```

---

sleep (Windows only)

`sleep <msecs>`

Pause the execution and sleep for <msecs> milliseconds.

Example 214 ----------

```
sleep 1000
```

---

step

`step [<Test Case or Test Group>]`

Start execution of the Test Case or Test Group and pause it after 1 step. When Test Case is running and paused, may be used without arguments to make 1 step in Test Case execution and pause it again.

Example 215 ----------

```
step TC
```

---

stop

`stop`

Pause Test Case execution.

timers

`timers`

List the available timers.

Files Handled by the SDL and TTCN Integrated Simulator

A total of 7 file types are handled by the SDL and TTCN Integrated Simulator user interface. Fortunately, most sessions involve only a subset of those file types. The file types and a short description are listed below:

- ETS file

  Executable Test Suite file. This is a file generated by the TTCN to C compiler. It might in particular be an integrated-simulator which has a specialized kernel for running in cooperation with an SDL Suite simulated system.

- Setup file (`.tts`)

  A Setup file is a collection of the other file types and is used to manage a particular test Setup such that it can easily be re-created.

- Log file (`.log`)

  A Conformance Log File consists of the output from an ETS, as specified in the ISO-9646 standard.

- Trace file (`.ttt`)

  A simplified version of the Log file, more corresponding to the actual TTCN tables contents.

- Test Suite Parameters file (`.ttp`)

  A file for transferring the test suite parameters to the ETS. Comparable to the PICS/PIXIT file of ISO-9646. For details, please see "Test Suite Parameters" on page 1347 in chapter 34, *The TTCN Exerciser*.

- Configuration file (`.ttc`)

  Test platform dependent file. Please see the ETS kernel implementators user guide for details.

- Breakpoints file (`.ttb`)

  A given set of breakpoints might be saved in the editor for later reuse.

Information Messages

The SDL and TTCN Integrated Simulator can present messages to the user in three forms:

- Informative messages
- Warning messages
- Error messages

Informative Messages

Informative messages are normally presented at the status bar. These might be for instance progress reports or other simple messages, which are only of a temporary interest (such as the tool tips).

Warnings Messages

Warnings, also known as “non-critical errors”, are events that indicate a problem during the preparation for an integrated-simulation, or during the actual integrated-simulation run. These are always presented in standard Motif warning dialogs. Warnings normally does not cause an integrated-simulation to halt.

Error Messages

Critical errors will halt the integrated-simulation. They are presented by an error dialog that appears in front of the application. The error dialog can often appear large and intimidating, but it simply gives a more verbose reason to why the error occurred, and what steps should be taken to avoid it.

Using the UI with an ETS

The user interface uses a publicly available format for communication with the ETS. Implementation of a management interface for an ETS is simple since the ETS only uses the standard I/O channels for communication with the GUI. Please request a protocol definition from IBM Rational if you consider adapting your own test suite using this GUI. It is currently unlicensed and might be started using this command:

```
$telelogic/itex/bin/${hosttype}bin/isimui -help
```

Type Mappings in Integrated-Simulation

This section specifies the data type mapping used by the integrated-simulation of the SDL Simulator and the SDL and TTCN Integrated Simulator. Specifically it identifies the transfer syntax used in the communication between the SDL Simulator and the SDL and TTCN Integrated Simulators for each supported data type.

TTCN Types

This subsection describes the mapping from TTCN types to SDL types. For each TTCN type the corresponding SDL type and some examples of the transfer syntax is given.

Predefined Types

TTCN Type	SDL Type	Transfer Syntax Example
INTEGER	integer	1 -3
BOOLEAN	boolean	true false
BITSTRING	bit_string	'0101'B
HEXSTRING	-	
OCTETSTRING	octet_string	'4BA0'O
NumericString	NumericString	'3295' (0..9 + space)
PrintableString	PrintableString	'ask38-'
TeletexString	-	
VideotextString	-	
VisibleString	VisibleString	'xy'
IA5String	charstring	'123 abc'
GraphicString	-	
GeneralString	-	

References Types

TTCN Type	SDL Type	Transfer Syntax Example
SimpleType	syntype	same as referenced type
Struct	struct	(. 1, true .)
PDU	struct	(. 1, true .)
ASP	- (ASPs maps to signals)	

ASN.1 Types

This subsection describes the mapping from ASN.1 types to SDL types. For each ASN.1 type the corresponding SDL type and some examples of the transfer syntax is given.

ASN.1 Type	SDL Type	Transfer Syntax Example
BOOLEAN	boolean	true false
INTEGER	integer	1 0 -55
ENUMERATED	enumeration types	The enumerated values
REAL	- (Not supported in the TTCN to C compiler)	
BIT STRING	bit_string	‘0100’B
OCTET STRING	octet_string	‘4BA0’O
NULL	null	Null
SEQUENCE	struct	(. 1, true .)
SEQUENCE OF	array, string	(: 1, 2 :)
SET	-	
SET OF	bag, powerset	[1, 2, 2]
CHOICE	choice	<name> : <value>
ANY	-	
OBJECT IDENTIFIER	-	

ASN.1 Type	SDL Type	Transfer Syntax Example
Selection type	-	
Tagged type	-	
SubType	syntype	same as referenced type

SDL Types

This subsection defines the mapping from SDL to TTCN and to ASN.1. For each SDL type first the TTCN type is given, then the ASN.1 type and finally an example of transfer syntax.

Predefined Sorts

SDL Type	TTCN Type	ASN.1 Type	Transfer Syntax Example
Bit	BITSTRING	BIT STRING (SIZE (1))	'1'B
Bit_String	BITSTRING	BIT STRING	'1010'B
Boolean	BOOLEAN	BOOLEAN	true false
Character	IA5String	IA5String (SIZE (1))	'a'
Charstring	IA5String	IA5String	'123 abc'
IA5String	IA5String	IA5String	'123 abc'
NumericString	NumericString	NumericString	'123'
PrintableString	PrintableString	PrintableString	'ask38-'
VisibleString	VisibleString	VisibleString	'xy'
Duration	-	-	
Time	-	-	
Integer	INTEGER	INTEGER	0 4 -66
Natural	INTEGER	INTEGER	0 55
Null	-	NULL	Null

SDL Type	TTCN Type	ASN.1 Type	Transfer Syntax Example
Object_Identifier	-	-	
Octet	OCTETSTRING	OCTET STRING (SIZE (1))	'FE'O
Octet_String	OCTETSTRING	OCTET STRING	'F0E2'O
PId	-	-	
Real	-	-	

User Defined Sorts

SDL Type	TTCN Type	ASN.1 Type	Transfer Syntax Example
Syntypes	SimpleTypes	SubType	same as referenced type
Enumeration Sorts	-	ENUMERATED	the enumerated values
Struct	struct	SEQUENCE	(. 1, true .)
Bit Fields	-	-	(. 2, 3, 0 .)
Optional	-	-	
Choice	-	CHOICE	C1 : 1

Predefined Generators

SDL Type	TTCN Type	ASN.1 Type	Transfer Syntax Example
Array	-	SEQUENCE OF	(: 1, 4, 7 :)
String	-	SEQUENCE OF	(: 1, 4, 7 :)
Powerset	-	SET OF	[1, 6, 8]
Bag	-	SET OF	[1, 6, 6]

Chapter 28 The SDL and TTCN Integrated Simulator (U)

Chapter

29

Customizing the TTCN Suite (on UNIX)

This chapter describes how to customize the TTCN Suite on UNIX. You can, for example, find information about how resources are read and how to customize key and button bindings.

The chapter does not contain information about how to configure the help environment. You can read about that in "Customizing the On-Line Help" on page 298 in chapter 4, *Managing Preferences*.

> **Note: UNIX version**
>
> **It is only possible to customize the TTCN Suite on UNIX with the facilities described in this chapter.**

Customizing the TTCN Suite

The TTCN Suite is an Xt based application (X11R5). Customizing is done by setting resources in the normal way for Xt applications. The default configuration is given in the file `$telelogic/X11/app-defaults/Itex`[1]. This file should be installed as part of the installation of the TTCN Suite.

Most of the resources in the global resource file for the TTCN Suite should never be overridden.

The resources that are most likely to be changed, will be mentioned in this chapter. For examples on how to set the resources see the file `Itex.sample` in the installation directory `$telelogic/itex`.

Note: UNIX only

Customizing the TTCN Suite is only possible **on UNIX**.

How Resources Are Read

At start-up, resources are read from a number of files in the standard way for Xt applications. There are a number of environment variables that control the way the program looks for resource files:

- `XFILESEARCHPATH`
- `XUSERFILESEARCHPATH`
- `XAPPLRESDIR`
- `XENVIRONMENT`

When the TTCN Suite is started, none of the environment variables above need to be set. The TTCN Suite will automatically find the resource file under the default name `$telelogic/itex/X11lib/app-defaults/Itex`.

To override some of the resources in the global resource file there are several possibilities:

- Give command line arguments to the TTCN Suite (see the man page `itex`).

1. `$telelogic` denotes the installation directory on your system.

- Set the environment variable `XENVIRONMENT` to a file containing resources for the TTCN Suite, or create a file `$HOME/.Xdefaults-<host>`.
- Set the value of the `RESOURCE_MANAGER` property with the program `xrdb`, or create a file `$HOME/.Xdefaults`.

The easiest way to set some personal resources is to create a file `.Xdefaults` in the home directory and add these resources to it.

For a more detailed description of how resources are found in an Xt based application see some suitable book on the subject, for example the *X Toolkit Intrinsics Programming Manual* from O'Reilly & Associates.

Customizing Key and Button Bindings

Key bindings are accomplished by translations in the resource file of the TTCN Suite. The translation tables that are relevant to add/change in are:

Itex*XmText.translations

Affect all places where text is edited in the TTCN Suite, for example rename of node in Browser and editing of field in the Table Editor. It is here the Emacs-like editing keys are defined.

Itex*textWindow*XmText.translations

Affect only key bindings in log windows. This is, by default, defined to contain the same key bindings as above but with the addition of a few bindings that have to do with scrolling.

Itex.browser*node.translations

Most of the keys in the Browser are defined here.

Itex.nodeTranslations

Some keys for traversing the Browser tree are defined here.

Itex.editor.headerFields.overrideTranslations

Defines the bindings available when focus is on a header field in the Table Editor.

Itex.editor.rowFields.overrideTranslations

Defines the bindings available when focus is on a body field in the Table Editor.

Itex.editor.fieldEditor.overrideTranslations

Defines editor specific bindings when editing a field in the Table Editor.

In the resource file there are comments that describe the action functions that can be used in translation tables. Changes to these tables must be done with caution, since much of the functionality of the TTCN Suite depends on these translations being set in a reasonable way.

Other Customizations

The following is a list of miscellaneous customizations. For examples, see the file `Itex.sample` in the installation directory `$telelogic/itex`.

Change of Paper Size

By default, the printout will be in A4 format. Other formats supported are Letter, Tabloid and Legal. For all formats it is possible to choose between portrait and landscape mode. The default is portrait mode.
See the file `Itex.sample` for an example.

Change of the Header/Footer in Printouts

By default, the header consists of a string containing the version, the document name and the string “user name @ host name”. It is possible to change the header and the footer.
See the file `Itex.sample` for an example.

Change of Font Size in Editor

By default, the font size used in fields in the Table Editor is 10 screen points. The same size is used in printouts. It is possible to change the

size of the Table Editor on the screen, without changing the paper size. See the file `Itex.sample` for an example.

Change of Font Family

By default, all text is from the Helvetica family. You can change this to another family supported by both X11 and Postscript (for example Times).
See the file `Itex.sample` for an example.

Changing Relative Widths of Columns in Editor

The total width of a table is divided among all columns. You can control this by specifying how wide different columns should be. It can be specified individually for each table type.
See the file `Itex.sample` for an example.

Hiding and Showing the Browser Toolbar

If you set the resource entry `Itex*browser.enableToolBar` to False, it is possible to hide the tool bar in the Browser window.

Configuring the Help Environment

The Help environment is configured via the Preference Manager, see "Customizing the On-Line Help" on page 298 in chapter 4, *Managing Preferences*.

Chapter 29 Customizing the TTCN Suite (on UNIX)

Chapter

30

Editing TTCN Documents (in Windows)

In the TTCN Suite, TTCN test components can be edited on table level as well as restructured on a hierarchical level. This chapter describes the editors and information managers in the TTCN Suite in Windows and how to use them.

For an overview of the TTCN Suite, see chapter 2, *Introduction to the TTCN Suite (in Windows), in the TTCN Suite 6.2 Getting Started*.

> **Note: Windows version**
>
> **This chapter is Windows only. The corresponding UNIX chapters are chapter 24, *The TTCN Browser (on UNIX)* and chapter 25, *The TTCN Table Editor (on UNIX)*.**

Introduction to the TTCN Suite

The TTCN Suite can be used for development, specification and compilation of test system components in the TTCN language. The functionality included will be described below.

The TTCN Suite supports all of the standardized, non-compact tables that are defined in ISO/IEC 9646-3. The compact test case dynamic behaviour table (ISO/IEC 9646-3, clause C.3) and concurrent TTCN tables are also supported. The compact tables for constraints (also in Annex C of ISO/IEC 9646-3) are not supported.

The general user interface concepts and common menu choices are described in chapter 1, *User Interface and Basic Operations*.

Different Views of the TTCN Document

The contents of a TTCN test suite can be viewed in *the Browser*, *the Finder* and *the Table Editor*:

- The Browser presents an overview of the TTCN tables. In the Browser it is possible to edit the TTCN table structure, to determine which parts of the test system component to view and to apply a number of operations on it. See "Using the Browser" on page 1253.
- The Table Editor presents TTCN tables for editing, referencing, and a number of table-specific operations. See "Editing Tables" on page 1262.
- The Finder presents a non-hierarchical view of the TTCN tables. Subsets of the total set of tables can be extracted, sorted according to various rules, and a number of operations can be performed on the resulting set of tables. See "Finding and Sorting Tables" on page 1282.

The Log Manager gives another view of the test suite. It presents log outputs from the currently loaded test system components. Information from various operations, such as analysis and simulation, will appear in the test component log in the Log Manager. See "Viewing Log Information" on page 1279.

Functionality to Apply on the TTCN Document

Building Test Components and Tables

- The Data Dictionary is used in conjunction with the Table Editor, and offers an easy way to build TTCN behavior statements from the declared test system components. For more information, see "Creating Behaviour Lines" on page 1269.
- SDL to TTCN Link is similar in use and appearance to the Data Dictionary. It utilizes a Link executable for an SDL system to interactively build the behavior of a TTCN test for that particular system. For more information, see chapter 35, *TTCN Test Suite Generation*.
- Autolink supports semi-automatic generation of TTCN test suites based on SDL specifications. For more information, see chapter 35, *TTCN Test Suite Generation*. The menu choice *Autolink Merge* in the *File* menu may be used for merging Autolink generated MP files into an opened test suite.

Analyzing, Verifying and Executing a Test

- The Analyzer verifies syntactic correctness of test components. For more information, see chapter 31, *Analyzing TTCN Documents (in Windows)*.
- The TTCN to C compiler generates C code from TTCN. For more information, see chapter 32, *The TTCN to C Compiler (in Windows)*.

  The Generic Compiler Interpreter interface is an interface for the adaption of the generated code to a specific target environment. For more information, see chapter 36, *Adaptation of Generated Code*.
- The Simulator allows execution and interactive debugging of test systems. For more information, see chapter 33, *The SDL and TTCN Integrated Simulator (W)*.

Starting the TTCN Suite

You can start the TTCN Suite from the Organizer in the following ways:

- Select *Tools > Editors > TTCN Browser* in the Organizer. This will also add an untitled TTCN document in the Organizer.
- Double-click on a TTCN document icon already included and connected in the Organizer (or select *Edit* from the *Edit* menu).
- Add a new TTCN document to the Organizer and make sure that the option *Show in editor* is checked in the *Add New* dialog.
- Add an existing TTCN document to the Organizer.

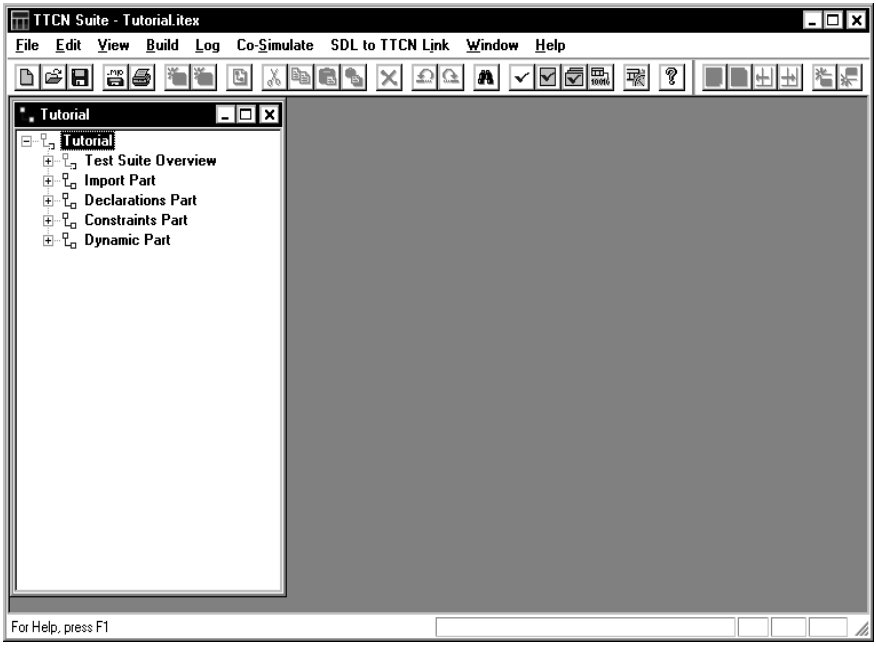


Figure 216:The TTCN Suite is started and the Browser displays a modular test suite

Using the Browser

When you open a TTCN document it will be displayed in the Browser. As the structure of TTCN is tree oriented, the Browser presents an overview of the test system component in tree form.

Multiple Browser views can be opened on a document and all editing performed in a Browser will be simultaneously reflected in the other opened Browser windows.

In addition to this, you may create sub Browsers. This means that you restrict the view to only display a few nodes. In combination with the possibility to have multiple Browsers open on the same test document, this enables you to keep just the interesting bits of information in a couple of small Browser windows.

By using the edit operations, such as *Add In* and *Delete*, you can manipulate the Browser structure to build a complete TTCN system component. You can control the amount of information actually displayed in a Browser, by collapsing and expanding sub trees.

Opening the Browser and a TTCN Document

You open the Browser and a TTCN document by selecting *New* or *Open* from the *File* menu. From the *File* menu, you can also select the four most recently used files.

> **Hint:**
>
> If you try to open very large test suites, for example containing extensive constraint declarations, you may get an error message about “scanner buffer overflow”. You can solve this by setting the environment variable ITEX_SCANNER_BUFFER_SIZE to a high value, for example 75 000 000.

To open additional Browsers for the same component:

- Make sure the Browser is active and select *File* > *New Window*.

The Browser is also opened automatically when the TTCN Suite is started from the Organizer, as described in “Starting the TTCN Suite” on page 1252.

If you try to open a TTCN document that is already opened by any another user or in another TTCN Suite Browser, you will get a question

informing that the file is opened by another user and ask to open it in read-only mode. In read-only mode you will not be able to save your changes (if any) in the same file, but free to save the TTCN document as another file. No other restrictions exist in read-only mode: you may browse, analyze, generate code, etc.

The question that the file is opened by another user is isuued because a read-lock file (with additional suffix '-read_lock') exists in the same directory as the document. This may also happen if the TTCN Suite or Windows has crashed. In this case just remove this file and try to open the TTCN document again.

The Browser User Interface

The initial structure of a TTCN tree contains place holders for the static parts of the system component. These *static nodes* are automatically created by the TTCN Suite whenever you create a new test system component. What you usually do when you build a test system component is to add *dynamic child nodes* to the static nodes, and fill them with test specification data. Examples of static nodes are the Declarations Part and the Constraints Part. Examples of dynamic nodes include tables, groups or objects in a table (e.g. test case variables).

The Browser displays all the static nodes in the order defined in the TTCN standard. This ordering cannot be changed. For example, it is not possible to have the Constraints Part coming before the Declarations Part. Only the dynamic items – the tables – may be added and deleted.

Static nodes are the only nodes that can become the root of a sub Browser.

Static Nodes in the Browser

Static nodes can take on four different appearances depending on the *parse status* of dynamic nodes below. Whenever you analyze a sub-tree, or parts of it, or edit it in a way such that the parse status of one or more dynamic nodes is altered, an abnormal parse status is indicated by the appearance of the static parent nodes.

This is an ordinary static node. All children to this node have been analyzed with status OK.

This is a static node where at least one of the dynamic nodes in the sub tree below has not yet been analyzed or is in need of analysis. There are no erroneous dynamic nodes.

This is a static node where at least one of the dynamic nodes in the sub tree below has been analyzed with an error or warning as result. In the actual view, the arrow is **red** for errors and **purple** for warnings.

To find a table that has to be corrected, you follow the marked nodes through the tree. Note that erroneous nodes have precedence over non-parsed ones, so even a single erroneous child to a static node will make it red no matter how many non-analyzed nodes there are in the sub-tree.

It may be that some of the static nodes will remain unused. A TTCN test system component is allowed to leave out the use of for example *Test Case Variables*. However, although this empty node will be displayed in the Browser, it will not be printed or output in the TTCN-MP format.

Dynamic Nodes in the Browser

A dynamic node is always associated with a table and can be accessed by the Table Editor. Two kinds of dynamic nodes can have children, namely *multiple tables* and *group tables.* Multiple tables look just like ordinary table icons, with the added feature of possible children, whereas group tables (referred to as *group nodes* in the Browser) have a different look to distinguish them somewhat from the others.

This is a group node with all dynamic nodes in its sub tree analyzed with status OK.

This is a group node with a sub-tree containing at least one non-analyzed dynamic node, but no erroneous ones.

This is a group node with a sub-tree containing at least one dynamic node analyzed with an erroneous status or a warning. In the actual view, the color is red for errors and purple for warnings.

This is a dynamic node (or multi-table) that is analyzed with status OK.

This is a dynamic node that needs to be analyzed.

This is a dynamic node that has been analyzed and contains one error. In the actual view, the cross is red for errors and purple for warnings.

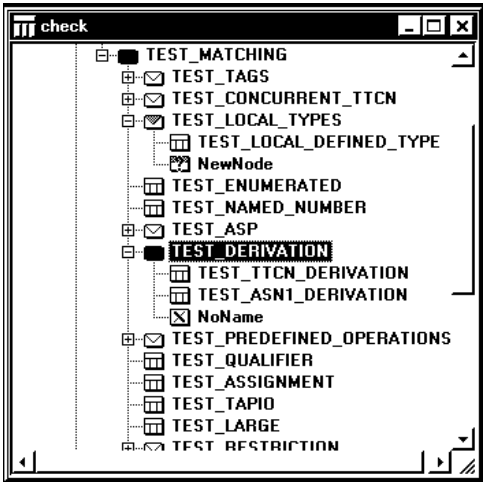


Figure 217: Different node types in a Browser

Controlling the Nodes in the Browser

As the Browser displays the information in a TTCN document in a tree oriented way, it is very simple to control the amount of information that the you want to see at a specific moment.

Expanding and Collapsing a Node

Nodes with sub-trees are possible to expand and collapse to show or hide the branch below it. This works in the same way that you expand and collapse directory levels in the Windows Explorer by double-clicking.[1]

- The shortcut for collapse is `<Left arrow>` when an expanded node is selected.

1. Note that test group nodes differ. Double-clicking a group node that is expanded always opens its associated table instead of collapsing it. Click on the minus sign to the left of the group icon instead.

- The shortcut for expand is <Right arrow> when a collapsed node is selected.

Selecting Nodes

Click a node to select it. <Ctrl>-click a node to extend the current selection.

Instead of using the mouse for navigation, you can use the following shortcuts:

- To move the selection to the next node, use <Down Arrow>.
- To move the selection to the previous node, use <Up Arrow>.

Renaming Dynamic Nodes

To rename a dynamic node:

- Click the name of a selected node, then type the new name and press <Enter>.
  – You can also change the name in the table itself.

Sorting Dynamic Nodes

The Sort command is applicable for any node that has dynamic subnodes. The command sorts the sub-nodes in alphabetical order.

To sort a dynamic node do one of the following:

- Right-click a node, then choose *Sort* from the shortcut menu.
- Select the node to sort. From the *Edit* menu choose *Sort.*

Creating a Sub Browser

To create a sub Browser:

- Select *Make Sub Browser* from the *View* menu. This replaces the contents of the active Browser with the selected node and its sub tree.

To get a Browser with the full TTCN document tree, select *New Window* from the *Window* menu.

Editing the Browser Structure

The static structure is automatically generated when a TTCN document is first created and cannot be edited. To build an individual TTCN system component, you need to add named tables – such as PDUs, constraints and behavior tables – and named objects to the multi-object tables – such as test case variables, PCOs and timers.

Use the *Edit* menu choices for adding, deleting, copying, pasting, etc, *editable nodes* – also called dynamic nodes – (that is, groups, tables or objects in a multiple object table) in the Browser.

Generating the Test Suite Overview

The test suite overview is a collection of tables that contain test suite structure, test case index and default index. These are used for reference in a printed copy of the system component, as described in the TTCN standard. The overview will be generated the first time you print the test suite, export it or open the overview tables. The generation may take a little while, but it is only needed once for each session, as the tables are automatically updated when you edit the test suite.

Adding Nodes

To add a new dynamic node as the last child of the selected node:

- Select *Add In* from the *Edit* menu.
  – The shortcut is `<Ctrl+Shift+A>`.

  In the case of multiple-object tables, select the node that represents the multiple-table. When you select *Add In*, a new node will be added as the last child node.

To add a new dynamic node before the selected node:

- Select *Add* from the *Edit* menu. As with *Add In*, the new table will be unnamed.
  – The shortcut is `<Ctrl+A>`.

Adding Groups

It is possible to add a new test case group, a new test step group or a new default group in the Dynamic Part, it is also possible to add a new Group for constraint declaration in the Constraints Part.

To add a new group as the last child to the selected node:

- Select *Add Group In* from the *Edit* menu.

To add a new group before the selected node:

- Select *Add Group* from the *Edit* menu.

To add a table to a group, select the group and *Add In*.

Adding Compact Tables

It is possible to specify that all the test cases in a given group are displayed in the compact format (see Annex C, clause C.3 of ISO/IEC 9646-3). The following commands allow the insertion of compact groups in the test suite hierarchy:

To add a compact group before the selected node:

- Select *Add Compact Group* from the *Edit* menu.

To add a compact group as the last child of the selected node:

- Select *Add Compact Group In* from the *Edit* menu.

Cutting, Copying and Pasting Nodes

You may cut or copy dynamic nodes. Dynamic nodes may also be pasted according to compatible classes, for example:

- Test suite parameters, test suite constants, test suite variables and test case variables may be pasted into each other.
- TTCN ASP, PDU, structured type and CM definitions may be pasted into each other and into TTCN ASP, PDU, structured type and CM constraints (and vice versa).
- ASN.1 ASP, PDU, structured type and CM definitions may be pasted into each other and into ASN.1 ASP, PDU, structured type and CM constraints (and vice versa).
- Test case, test step and default behaviors may be pasted into each other.

However, it is not possible to paste a constraint, for example, as a test case behaviour.

Paste is not available if the clipboard contains an object of an incompatible type to the selected object.

To cut the selected node and its sub tree:

- Select *Edit* > *Cut*.
  – The shortcut is `<Ctrl+X>`.

To copy the selected node and its sub tree:

- Select *Edit* > *Copy*.
  – The shortcut is `<Ctrl+C>`.

To paste a node before the selected node:

- Select *Edit* > *Paste*.

To paste a node as the last child to the selected node:

- Select *Edit* > *Paste In*.
  – The shortcut is `<Ctrl+V>`.

Creating Constraints

The most convenient way of creating a constraint is to copy the corresponding type and paste it as a constraint. Do this by copying the chosen type (ASP, PDU, CM or structured type), and then pasting it among the corresponding constraints. This works for both tabular constraints and for ASN.1 constraints.

The pasted table will have all its local names filled in. All that remains is to give the constraint a name and fill in the values.

Deleting Nodes

Dynamic nodes may be removed from the test document.

To delete the selected node and its sub tree:

- Select *Delete* from the *Edit* menu.

Undoing/Redoing Operations

All editing commands can be undone, and undone actions can be re-done.

- To revert the last edit step, select *Undo* from the *Edit* menu.
  – The shortcut is `<Ctrl+Z>`.

- To revert the last *Undo*, select *Redo* from the *Edit* menu.
  – The shortcut is `<Ctrl+Y>`.

Opening a Table

To open a table, either:

- Double-click a table node in the Browser.
- Select a table node and then press `<Enter>`
- Select a table node and then *Open Table Editor* from the *View* menu.

It is possible to open several tables at the same time, but if it is more than ten, you will be asked to confirm it.

Printing a TTCN Document

To print a TTCN document:

- Select *Print* from the *File* menu.
  – The shortcut is `<Ctrl+P>`.

For more information, see "The Print Dialogs in the TTCN Suite" on page 333 in chapter 5, *Printing Documents and Diagrams*.

Editing Tables

The Table Editor is used for editing various TTCN tables. It displays the tables in three different parts: the header, the body and the footer. Also, in the upper part, it displays the Search and Replace tool. Each part, as well as many fields, can be resized.The Table Editor is used for editing various TTCN tables. It displays the tables in three different parts: the header, the body and the footer. Each part, as well as many fields, can be resized.

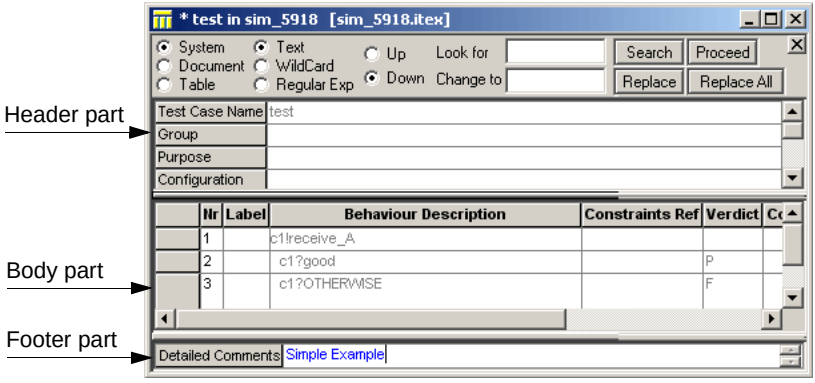


Figure 218: A Table Editor for a test case

Resizing Cells and Table Parts

The three parts of a table – header, body and footer – are separated by horizontal bars, which you can drag to change the relative size and to hide parts of the table from view.

You can change the individual height of rows by dragging the leftmost row separator. In the body part, you may also change the individual width of columns by dragging the separators of the header fields.

Note that changing the column width may have effect on the row height, as the row will change to display all information it contains.

When the Search and Replace tool is closed this affects all Table Editors opened in TTCN Suite. To restore the view of Search and Replace tool press the expand button that appear in upper-right part of Table Editor instead of the **Close Window** button.

Renaming a Table

To change the name of a table, you can either rename its node in the Browser or edit the table name in the table itself.

Setting the Input Focus

To set the input focus in another part of a table, either:

- Click in a cell in that part.
- Use `<left arrow>` and `<right arrow>` to move between the last field of one part and the first field of another part.

To set the input focus within a part, either point and click or use the arrow keys. There are also some shortcuts:

To Move to:	Use This Key Combination
Next field in same row	`<Tab>`
Previous field in same row	`<Shift+Tab>`
Last field in same column	`<Ctrl+Down arrow>`
First field in same column	`<Ctrl+Up arrow>`
First field	`<Ctrl+Home>`
Last field	`<Ctrl+End>`

Editing Text in a Table

When you have used the keyboard to move the input focus to a field, the contents of the field will be **replaced** with the new text when you start typing. To edit the existing text, you have to press `<Home>` or `<End>` first.

If you use the mouse to explicitly set the input focus in a field, you can start editing the existing text directly with the keyboard.

The table fields will expand automatically to accommodate the text. You can also add a line break in the text by using `<Return>`.

To cut, copy and paste text in the fields, you use the corresponding commands in the popup menu, the *Edit* menu or the standard Windows

shortcuts. However, note that it is only possible to paste text into a table field when the field contains a text pointer.

> **Note:**
> The paste buffer for text is not the same as the one used for entire rows in the body of a table.

Auto completion of identifiers

When editing tables, it is possible to auto-complete identifier table names while editing text in the Table Editor.

Start to type any identifier (a table with this name should already exist in the test suite). Type one or more letters, then press <CTRL-space>. TTCN Suite will find the tables that match the text typed. If one match is found, the typed name will be completed with the found one. If several matches are found, a pop-up window with a list of names will be displayed. It is possible to select one by using keyboard (“Up”, “Down” arrows for navigate, “Enter” to select, “ESC” for cancel) or mouse (double-click to select, click outside the window for cancel).

Editing Rows in the Body of a Table

You can add, delete, cut and copy rows in the body of all TTCN tables that contain more than one column. This is not possible in the headers and footers, since the formats of these parts of a table are defined by the TTCN standard.

The ASN.1 tables only have a single column with a single row and therefore adding, deleting, cutting and copying rows is not applicable. However, the contents of ASN.1 tables can still be copied, pasted, etc, as text.

Selecting and Deselecting Rows

- Select rows in the body of a table by clicking in the leftmost cell of a row (the “row button”). This deselects other selected rows.
- Select range of rows in the body by first selecting one row and then `<Shift>`-clicking another.
- Toggle the selection of a row by `<Ctrl>`-clicking the leftmost cell of the row.

Note:

Setting the input focus will also deselect all selected rows.

Cutting, Copying and Pasting Rows

Rows in the body of a table may be cut, copied and pasted. *Copy* and *Paste* work across all the different tables although the *Paste* command is mainly intended for use among tables of the same or similar types.

To cut selected row or rows from the table:

- Select *Edit* > *Cut*.
  - The shortcut is `<Ctrl+X>`.

To copy selected row or rows:

- Select *Edit* > *Copy*.
  - The shortcut is `<Ctrl+C>`.

To paste a row or rows:

- Select *Edit* > *Paste*.
  - The shortcut is `<Ctrl+V>`.

To paste a row or rows before the current row:

- Select *Paste In* from the *Edit* menu.

It is possible to paste a row when a body row or field is selected.

Note:

The paste buffer for entire rows in the body of a table, is not the same as the one used for text.

Deleting rows

To delete the selected row or rows, select *Delete* from the *Edit* menu.

Inserting Rows

To insert a new row before a selected row:

- Select *Insert New Row Before* from the *Edit* menu.

  Only one row may be selected. If no row is selected, the new row will be created first in the table.

  – The shortcut is `<Ctrl+Ins>`.

To insert a new row after a selected row:

- Select *Insert New Row After* from the *Edit* menu.

  Only one row may be selected. If no row is selected, the new row will be created last in the table.

  – The shortcut is `<Ins>`.

To insert a new tree header row before a selected row:

- Select *Insert Local Tree Before* from the *Edit* menu.

  Only one row may be selected. This command only works in dynamic behaviour tables. If no row is selected, the new row will be created first in the table.

To insert a new tree header row after a selected row:

- Select *Insert Local Tree After* from the *Edit* menu.

  Only one row may be selected. This command only works in dynamic behaviour tables. If no row is selected, the new row will be created last in the table.

You may also use `<Ins>` to insert rows. Where the row will be inserted depends on the input focus:

- If input focus is in the header or footer of a table, a new row is added after the last row in the body of the table.
- If input focus is set on a field in the body of the table, a new row is added after the field that has the input focus. The input focus will be transferred to the corresponding field in the new row.

- <Ctrl+Ins> gives the same effect as above but the new row is inserted **before** the row/field.

Indenting Behaviour Lines

The indention of behaviour lines can be increased and decreased:

To increase the indentation of a row in focus, or selected rows, by one position:

- Select *Increase Indention Level* from the *Edit* menu.

To decrease the indentation of row in focus, or the selected rows, by one position:

- Select *Decrease Indention Level* from the *Edit* menu.

Browsing in the Table Editor

If you right-click a table field containing an identifier, you may open the table representing the identifier from the pop-up menu:

- To save the current table in the history and display the definition of the selected identifier in the current Table Editor:
  - Select *Open <Identifier>* from the pop-up menu.
- To display a pop-up menu containing the header rows of the identified table:
  - <Control>-right-click the identifier.
- If the identifier is associated with a one-line table, the whole table is displayed in the pop-up. To select the identifier in the popup will have the same effect as *Open <Identifier>* (described above).
- To open the table associated with the identifier in a new Table Editor window:
  - Select *Open <Identifier> in New Window* from the popup menu.

    The new window gets a new (empty) history buffer and is completely independent of the table from which it was opened.

The Table Editor maintains a history of tables displayed with *Open <Identifier>* described above. This is similar to going back and forward in a web browser:

To go forward in the history of the Table Editor:

- Select *Go Forward* from the popup menu or the *View* menu.

To go back in the history of the Table Editor:

- Select *Go Back* from the popup menu or the *View* menu.

Editing Text with an External Editor

You may use any external text editor for editing text in TTCN Tables. The editor must have the possibility to read and save files in ASCII format (text only, without any formatting).

Specify the external editor executable in the TTCN Suite Options dialog (*Menu > Settings > Options...*). Right-click in any cell of TTCN Table, with no text selected in the cell, and point to the item *External Editor* in the shortcut menu. to start the editor which you specified. After you have edit the text, save and close the editor. This will import the modified text to the cell of TTCN Table. While the external editor is open no other operations in the TTCN Suite are available. The external editor must be closed before you continue.

Creating Behaviour Lines

The Data Dictionary gives you an alternative way of writing behaviour lines, by providing easy access to lists of PCOs, types, constraints, timers, etc, that you have already defined

Before you can use the Data Dictionary, the TTCN document has to be analyzed, because PCOs, types and constraints with major reference problems or missing type references will not be presented in the lists in the dialog.

Opening the Data Dictionary

To open the Data Dictionary:

1. Select a body row in a behaviour table.
   - If you have not inserted a behaviour line yet, the input focus should be somewhere in the header of the table.
2. Select *Data Dictionary* from the *View* menu.

It is only possible to have one *Data Dictionary* dialog opened.

The *Data Dictionary* dialog contains:

- Three different tabs: *Send/Receive*, *Timer* and *Attachment*, from which you can select different statements to generate.
- Three fields: *Behaviour Line*, *Constraint* and *Verdict*, whose contents will be inserted as a TTCN line in the table. It is also possible to edit the fields manually.
- Three buttons: *Apply*, *Clear* and *Close* (the first two only enabled when a body row of a behaviour table is selected):
  - Click the *Apply* button to add a new row – containing the contents of the *Behaviour Line*, *Constraint* and *Verdict* fields – to the table.
  - Click the *Clear* button to reset the dialog and deselect all selections in the current tab.

Generating a Send or Receive Statement

Generate a send or receive statement by selecting a PCO, a type and a constraint. You can also add a timer, assignment and qualifier.

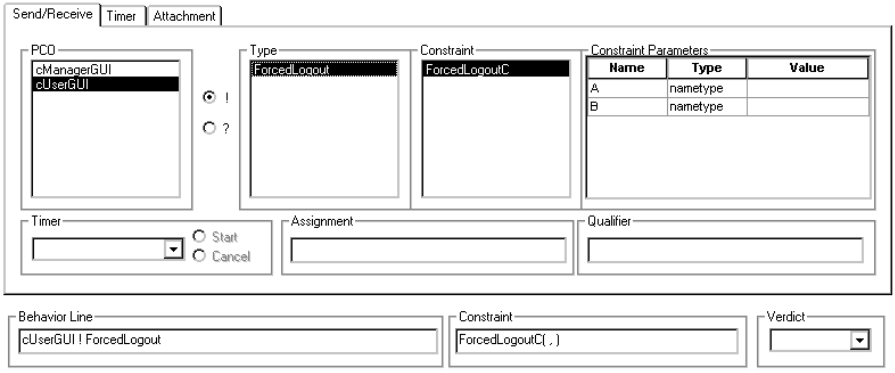


Figure 219: The Send/Receive tab in the Data Dictionary dialog

To add a send or receive statement:

1. Select the *Send/Receive* tab.
2. Select a PCO.

   The *Type* and *Constraints* list will be updated and only show the entries that are valid for the currently selected PCO.
3. Select the *!* radio button for a send statement or the *?* radio button for a receive statement.
4. Select an ASP or a PDU in the *Types* list.

   The *PCO* and *Constraints* lists will be updated accordingly.
5. Select a constraint.

   The *Type* and *Constraints* list will be updated.

   – If the constraint is a parametric constraint, a choice for each parameter will be displayed in the *Constraints Parameter* area, which you can select.

6. Optionally you can add a timer statement by selecting the *Start* or *Cancel* radio button.

   To insert a start statement, you also have to select a timer.

   It is not possible to select timeout or to specify a timeout value here. You have to do that in the *Timer* tab of the dialog.

7. Optionally, add assignments separated by commas and without parenthesis.

8. Optionally, add qualifier statements, without brackets.

9. Optionally, add a verdict.

10. Click *Apply*

    A new row – containing the contents of the *Behaviour Line*, *Constraint* and *Verdict* fields in the dialog – will be added after the currently selected row and the new row will be selected.

Adding a Timer Statement

In the *Timer* tab, you may generate a StartTimer, CancelTimer or a Timeout statement. As described above, you can also add timer statements from the *Send/Receive* tab. The difference is that from this tab you can also add Timeout value to the timer and you can add a timeout statement.

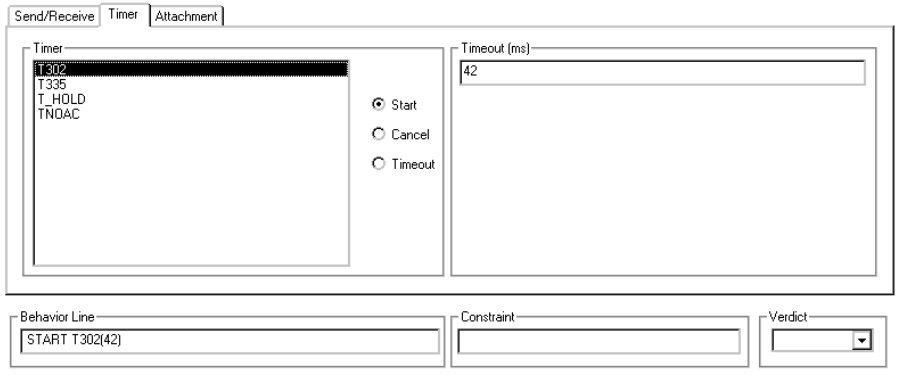


Figure 220: The Timer tab in the Data Dictionary dialog

To add a timer statement:

1. Select the *Timer* tab in the dialog.
2. Select a timer.
3. Select the *Start*, *Cancel* or *Timeout* radio button.
4. If you selected *Start* or *Timeout*, you also have to specify a timer duration in the *Timeout* field.
5. Click *Apply*.

   A new row – containing the contents of the *Behaviour Line*, *Constraint* and *Verdict* fields in the dialog – will be added after the currently selected row and the new row will be selected.

Adding an Attachment Statement

Generate an Attachment statement by selecting a test step and specifying an actual parameter list (if the test step has a formal parameter list).

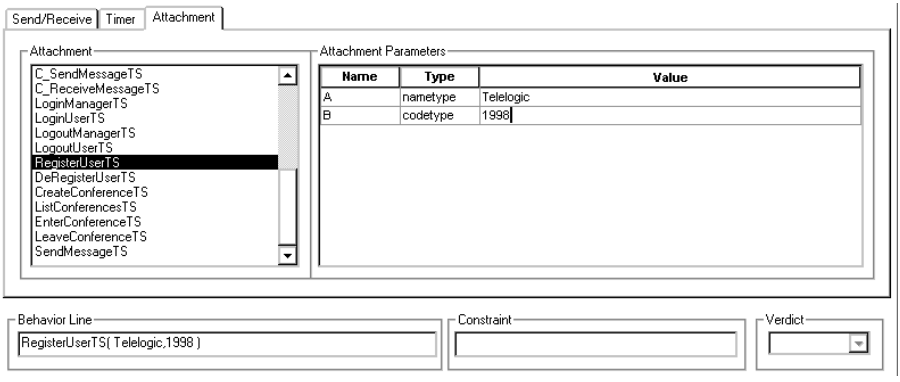


Figure 221: The Attachment tab in the Data Dictionary dialog

To add an attachment statement:

1. Select the *Attachment* tab.
2. Select an attachment.
3. If the selected test step has a formal parameter list, specify the parameter values in the *Attachment Parameters* area.

Comparing TTCN Documents

The Compare tool is used for comparing two TTCN documents with respect to finding *similarities* that exist between them, i.e. two TTCN objects that have the same name. These objects need not necessarily be of the same type.

The Compare tool is accessed from a Browser and is applied to selections of items in that Browser.

Compare will check either only for similar names or for names **and** object type. The comparison can also include the contents of tables if required.

The Compare tool can also check the group structure of TTCN documents for similarities, i.e. it will indicate Test Group, Test Step Group and Default Group paths that are the same in both TTCN documents.

Any similarities that are found may be logged in a log file. There are three levels of log verbosity: *quiet*, *some* and *full*. Execution of the Compare tool may result in a new selection, which can be used to create a separate Browser if so wished.

One of the main uses of this tool will be to help remove conflicts from two TTCN documents prior to performing a merge operation.

Comparing by Using the Compare Tool

To use the Compare tool, you need two TTCN documents to operate on: the *destination* document and the *compare* document.

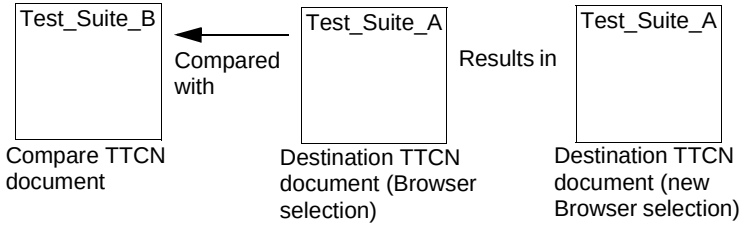


Figure 222: How Compare works

A selection from the *destination* document may be compared with the *compare* document. If any similarities are found between the two TTCN

documents this will result in the modification of the original selection in the *destination* TTCN document.

Open both the *destination* TTCN document and the *compare* TTCN document in browsers. Make the selection in the *destination* TTCN document Browser and call the Compare tool from the menu command ***File -> Compare*** (should this menu item be dimmed, most likely you have currently only one open TTCN document). The following Compare dialog will appear:

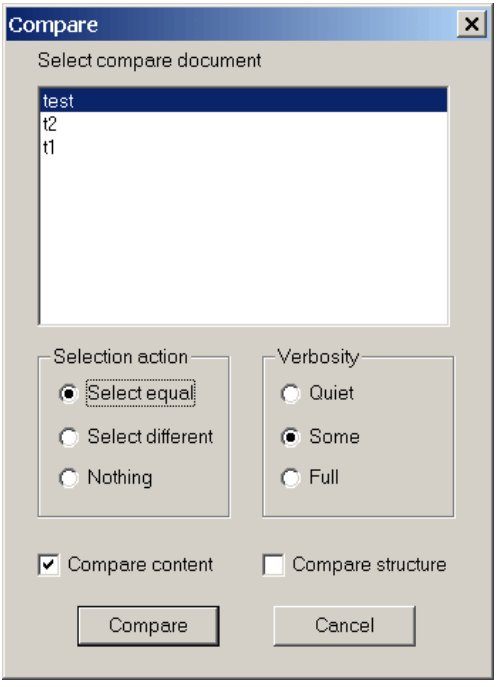


Figure 223: Compare Documents dialog

Select compare document

Presents a list of currently opened TTCN documents except the destination TTCN document, from which the Compare tool was called. Choose in the list the compare TTCN document.

Selection action

Specifies the effect that the Compare operation will have on the original selection in the destination TTCN document.

- *Select equal*

  All items in the original selection that are similar to items in the *compare* TTCN document will **remain selected**, otherwise the items will be deselected.

  In other words, the result of the Compare operation will result in a selection that shows the similarities between the *destination* and the *compare* TTCN documents.

- *Select different*

  All items in the original selection that are similar to items in the *compare* TTCN document will be **deselected**, otherwise the items will remain selected.

  In other words, the result of the Compare operation will result in a selection that shows the differences between the TTCN documents.

- *Nothing*

  Means that the Compare operation will have no effect on the destination selection. However, the results of the Compare operation will still be recorded in the log.

If the *Selection action* is anything except *Nothing*, the TTCN tree is being expanded up till all selected leaf items.

Verbosity

Sets the verbosity level for the log.

- *Quiet*

  Turns the log off. No information will be recorded in the log.

Note:

Using the *Nothing* action together with the *Quiet* option is rather meaningless!

- *Some*

Causes the names of all identically named objects to be printed in the log. If the *Compare content* switch is on, the log will also indicate that a difference has been detected.

- Full

  Causes the names of all identically named objects to be printed in the log. However, if the objects are of different types this will be indicated as well. If the *Compare Content* switch is on, the log will also indicate in detail the differences detected.

Compare Content

Extends the scope of the comparison to include the contents (down to the level of a single character) of the tables as well. Thus, two objects are considered to be identical if their names, object types and entire contents are identical. If the verbosity is full the differences are detailed.

Compare structure

Extends the scope of the comparison to include the Test Group, Test Step Group and Default Group references in the relevant behaviour tables. This option is only available when the Compare content option is chosen.

Merging TTCN Documents

The Merge tool is used for merging one TTCN document (complete or partial) into another TTCN document. There are two ways for merge:

1. Merge the selected tables from the open Test Suite into the another open Test Suite (*Merge to suite*).
2. Merge all tables from any Test Suite into the open Test Suite (*Merge from file*).

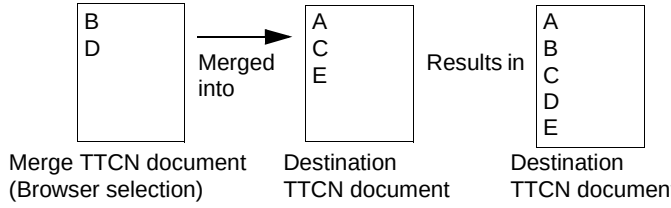


Figure 224: Shows how Merge works

Merge will only work as depicted above if the two TTCN documents do not conflict. A conflict occurs if any TTCN object in the *Merge TTCN document* has the same name as any TTCN object in the *Destination TTCN document*. If such a conflict is detected, the conflicting object will be skipped, information about this will be put in log, and the merge process continued.

Merging of constraint tables are handled separately. For example, the *Merge TTCN document* may contain a TTCN ASP Constraint, *constraint1*, that refers to the type *type1*, where the *type1* Type table is of the incompatible type *TTCN PDU TypeDef*. This will make the merge process insert a copy of constraint1 as a table of type *TTCN PDU Constraint* instead of the original type. There are, however, limitations to this type conversion, the conversion will for example not convert an ASN.1 Constraint to a TTCN Constraint nor vice versa.

Merge to suite

Open the *Merge TTCN document* and the *Destination TTCN document*. Select in *Merge TTCN document* the tables that should be merged into *Destination TTCN document*. From the *File* menu choose *Merge to suite* (should this menu item be dimmed, most likely you have currently

only one open TTCN document). A dialog with names of other open test suites will appear. Select the *Destination TTCN document* and press "Merge" button.

Merge from file

Open *Destination TTCN document*. From the *File* menu choose *Merge from file*. A file selection dialog will appear. Choose the file that contains the *Merge TTCN document*. GR (*.itex) and MP (*.mp) file formats are supported.

It is possible to select several files at same time. In this case they will all be merged one by one into the *Destination TTCN document*.

Merge from command line

`TTCNMerge` is a command-line tool for merging two or more TTCN documents into one.

Usage:

```
TTCNMerge <Destination TTCN Suite file> <Source file 1>
... <Source file n>
```

If `<Destination TTCN Suite file>` doesn't exist, it will be created.

Example 216 ----------

```
TTCNMerge total.itex main.mp decl.mp initial.itex
```

The appearance order of source files in command line is significant such that files are being merged one by one in order of their appearance, and, if the table with the same name exists in several TTCN source documents, only the first appearance will be merged. All others will be skipped with warnings given.

Viewing Log Information

The Log Manager is a collection of log outputs, where each test system component can report information. The Log Manager contains one log per test component, and one common pane that is used for general information.

Examples of actions that can result in log information:

- Analysis of one or more tables: The resulting errors (if any).
- Opening an MP file: Result of the operation.
- Simulation: Simulator traces are reported on both the component log, and the general log.

The Log Manager can be configured to behave in various ways. The default setting is for the Log Manager to be hidden, but appear whenever log output is present. Only one instance of the Log Manager is available.

The contents of each log can be saved as a text file.

You can open tables by clicking and then `<Ctrl>`-right-clicking the name of the table in the log.

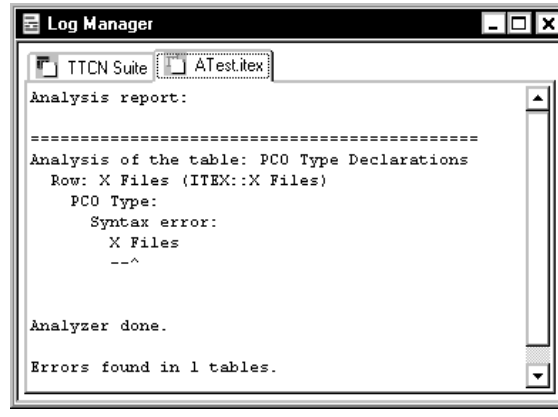


Figure 225: The Log Manager window

Automatic Appearance

The Log Manager can be made to automatically become visible whenever new information is available in a log. This can be handy to reduce the amount of windows open at a given time as the Log Manager can be closed when it is not needed.

To allow the Log Manager to automatically open whenever new information is available:

- Select *Auto-Raise Log* from the *Log* menu.

Changing the Appearance of the Log

The log font can be made smaller or larger, allowing more information to be viewed or increased readability.

To make the log text larger:

- Select *Increase Log Font Size* in the *Log* menu.
  - The shortcut is `<Ctrl+Alt+L>`

To make the log text smaller:

- Select *Decrease Log Font Size* in the *Log* menu.
  - The shortcut is `<Ctrl+Shift+L>`

Exporting Information from the Log

The information contained in a log can be copied to the clipboard, or saved to disk as an ordinary text file. In addition to normal text-selection, all text can also be selected with one operation.

To select all text:

- In the *Log* menu, point to *Select All In Log*.

To copy the selection to the clipboard:

- Select *Copy Log* in the *Log* menu.

To save all text in a log to a file:

- Select *Save Log* in the *Log* menu.

Clearing the Log

To clear the displayed log:

- Select *Clear Log* in the *Log* menu

Finding and Sorting Tables

As an alternative to the Browser, you can use the Finder for displaying TTCN tables in a list, without the ordering restrictions imposed by the TTCN tree structure.

You can search for and display tables in the Finder. The search may for example be based on name, type and content. You can use the entire TTCN document, entire TTCN Suite system, tables selected in TTCN Browser or the Finder Results List as search source, and extend, replace or restrict the number of tables displayed in Finder Results List in any number of steps.

Another way of searching is relationally. This means that you search for tables that reference or are referenced by tables selected in the Finder Results List or in TTCN Browser. The relational search cannot be combined with the ordinary.

The tables that will be displayed in Finder Results List after a search, can be sorted by a number of criteria:

- Name
- Type
- Path in the test tree
- Parse status
- Latest analyze time stamp
- Latest modification time

Reversed sorting is also possible.

In addition, it is possible to for example open, analyze, copy, rename, export to MP file and delete tables from the Finder Results List, but not cut, paste and insert since that require a tree context.

Searching and Replacing

The Search and Replace tool exists in the upper part of the Table Editor, see "Editing Tables" on page 1262 in chapter 30, *Editing TTCN Documents (in Windows)*. Its settings and functionality is absolutely the same as on UNIX, see "Searching and Replacing" on page 1176 in chapter 25, *The TTCN Table Editor (on UNIX)* for information on the TTCN Table Editor on UNIX. The difference is only that there are buttons instead of Search Menu items.

Opening the Finder

To open the Finder:

- Select *Open Finder* from the *View* menu.
  – The shortcut is `<Ctrl+F>`

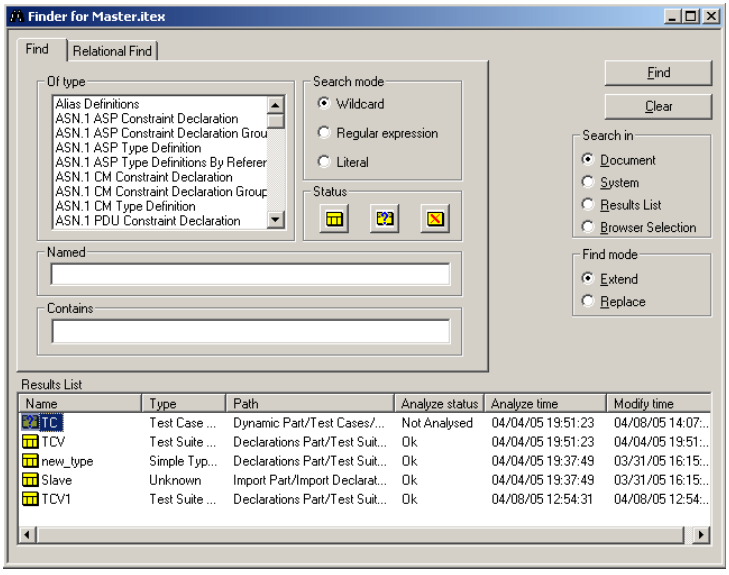


Figure 226: The Finder

The Finder consists of four areas:

- The *Find/Relational Find* area, where you specify criteria for the tables to be found and inserted in the *Results List*.
- The *Search In* area, where you can select to search for tables in the entire document, entire system, tables selected in Browser or only in the *Results List*.
- The *Find mode* area, where you can specify should the *Results List* keep previously found tables or not.
- The *Results List*, where the sorted list of TTCN tables are kept.

About Search Criteria

Search criteria are used for **restricting** a search. This means that if you do not specify any search criteria – all fields are empty and no buttons are clicked – before you click the *Find* button, all dynamic TTCN tables of the test system component will be displayed.

Consequently, if you specify one or more search criteria, TTCN tables that match at least one of the criteria, will be displayed in the Finder. Additionally, this means that specifying all possible search criteria has the same effect as specifying none. This is merely as a convenience for quickly adding all tables of the test system component, as the closed search criteria is not meaningful.

Note:

The search criteria are used for an OR-style search.

Finding Tables

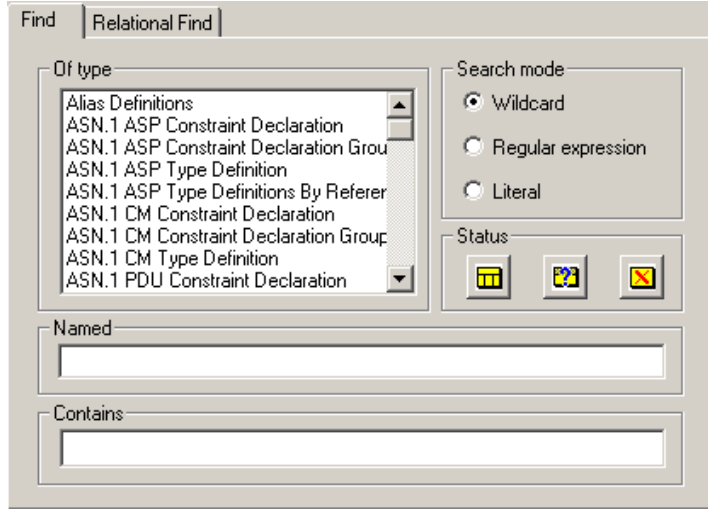


Figure 227: The Find tab in the Finder

To search for tables:

1. Select the *Find* tab.
2. Select Document, System, Results List or Browser Selection in the Search in area to search for tables in:
   - If you select *Document*, the search area will be the entire TTCN document.
   - If you select *System*, the search area will be the entire TTCN system.
   - If you select *Results List*, the search criteria will be used to restrict the set of TTCN tables already present in the *Results List*.
   - If you select *Browser Selection*, the search area will be the tables selected in the TTCN Browser.
3. Select Extend or Replace in the Find mode area to choose should Results List keep previously found tables or not.
4. Optionally, select one or several table types in the *Of type* list. No table types selected means that any table type appropriate.

   This list contains all existing TTCN and ASN.1 table types supported.

5. Click a *Search mode* radio button.

   This selection affects the text you type in the *Name* and *Contains* fields. If you are not going to use those fields in the search, the search mode does not affect the search at all.

   – *Wildcard*

     The *Name* and *Contains* fields can contain wildcards. This means that “a*body” will match “anybody” and “antibody”, while “a??body” only will match “anybody”.

   – *Regular expression*

     The *Name* and *Contains* fields contain regular expression, with its more advanced syntax and matching possibilities. This means that “any[bt]” will match “anybody” and “anything” but not “anyone”.

   – *Literal*

     The *Name* and *Contains* fields contain the exact text. This means that “any?” only will match “any?”.

6. Optionally, type some text in the *Name* field.

   The tables with names that match will be found.

7. Optionally, type some text in the *Contains* field.

   The tables with contents that match will be found.

8. Optionally, click one or several *Status* buttons to sort out tables of a different parse status.

   For example, if you select the error button, only tables with erroneous parse status will be found.

9. Click *Find*.

   Tables that match at least one of the search criteria you have specified will be displayed in the *Results List*.

Finding Tables Relationally

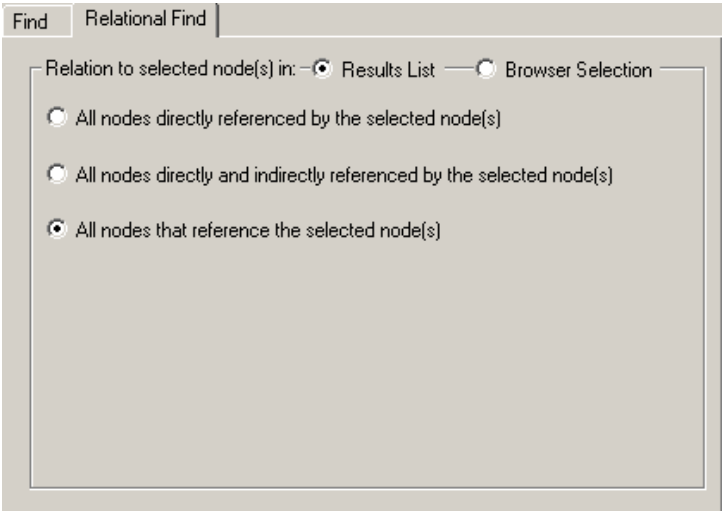


Figure 228: The Relational Find tab in the Finder

You can search for tables that are referenced by, and that reference tables, that are displayed and selected in the *Results list* or selected in TTCN Browser. It is required that the TTCN document is analyzed before start Relational Find. To do this:

1. Select the *Relational Find* tab.
2. Select one or several tables in the *Results List*.or in TTCN Browser and select one of the following radio buttons:
   - Results list
   - TTCN Browser
3. Select one of the following radio buttons:
   - *All nodes directly referenced by the selected node(s)*

     Replaces the contents of the *Results List* with all nodes that are directly referenced by the tables currently selected in the list.
   - *All nodes directly and indirectly referenced by the selected node(s)*

Same as above, but expands the search to include tables that are referenced by the referenced tables, and so forth.

– *All nodes that reference the selected node(s)*

Replaces the contents of the *Results List* with all nodes that reference the tables selected in the list.

4. Select Document, System, Results List or Browser Selection in the Search in area to search for tables in:
   - If you select *Document*, the search area will be the entire TTCN document.
   - If you select *System*, the search area will be the entire TTCN system.
   - If you select *Results List*, the search criteria will be used to restrict the set of TTCN tables already present in the Results List.
   - If you select *Browser Selection*, the search area will be the tables selected in TTCN Browser.
5. Select Extend or Replace in the Find mode area to choose should Results List keep previously found tables or not.
6. Click *Find*.

   The *Results List* will be updated in accordance with your settings.

Results List operations

The *Results List* contains a subset of all dynamic TTCN tables in the TTCN system. You can perform most standard operation (copy, delete, rename, open in Table Editor, export to MP, Analyze, etc) on the tables. However, it is not possible to cut, paste or insert tables, since this requires the structural context of the Browser.

Caution!

Even if the elements of the *Results List* are presented without structural ordering, they still follow the TTCN hierarchy. This means that if you delete a group node, the group and its entire sub tree will be removed.

Sorting the *Results List*

The contents of the *Results List* can be sorted if you click on the header of the information column. The available information is:

- *Name*

  The name of the table.

- *Type*

  The TTCN or ASN.1 type of the table.

- *Path*

  The path of the tree leading to this table on the form root/..../parent/node.

- *Analyze status*

  The parse status of the table.

- *Analyze time*

  The last time the table was analyzed.

- *Modify time*

  The last time the table was modified.

Click a second time on the header to sort the list in reverse order.

Clearing the *Results List*

Click *Clear* to clear the contents of the *Results List*.

When it is necessary to remove some individual tables from *Results List*, select them, right-click and choose *Remove* from *Results List* menu item.

Export to MP file

Tables from *Results List* may be exported to an MP file. To do this select *Tables*, right-click and choose *Export*.

How to isolate a single Test Case

A single Test Case may be isolated from a Test Suite together with all its dependent tables using the Finder.

To do this do the following:

- Analyze the Test Suite. The analysis must complete successfully without any errors.
- Select in the TTCN Browser window the Test Case that should be isolated (it may not be a single Test Case, and in fact even not a Test Case - you can select any table(s) that should be exported together with all their dependent tables).
- Start the Finder, open the "Relational Find" pane. In the section "Relations to selected node(s) in:" you select the "Browser Selections" and "All nodes directly and indirectly referenced by the selected node(s)". In the section "Search In:" you select "System", if the Test Suite is Modular. If the Test Suite is not modular, you can do the search in "Document".
- Press "Find" button. All tables that required by the selected in TTCN Browser window Test Case will appear in Results List.
- Select all these tables (click on the first table and then press Shift key and click on the last one), then right-click and choose "Export" in pop-up menu. Give the file name for the new file, that will contains all these tables.

In the case where you have a Modular Test Suite, the tables that belong to different modules will be exported to different files. Therefore the prompt to give new filenames will appear for each one of the exported

tables. As a result the exported Test Suite will also be Modular and the original structure will be kept. If it is necessary to convert it into a non-modular Test Suite, use the Generate Flat View option from the Analyze TTCN dialog, see "Analyze TTCN" on page 118 in chapter 2, *The Organizer*.

Converting to TTCN-MP

TTCN-GR – the *gr*aphical notation – is the format used when you edit or print a test suite in the TTCN Suite. TTCN-MP – the textual notation (*m*achine *p*rocessable) – can be used when you want to import a TTCN document into a non-TTCN Suite tool or make backups.

To convert a TTCN document to TTCN-MP, either:

- Select *Save As* from the *File* menu.

  This opens a dialog where you can save the current document as TTCN-MP.

- Click the *Export to MP* button.

  This opens a dialog where you can export the current document to TTCN-MP. The MP file will be written to disk, but not opened.

Both TTCN-MP and TTCN-GR can be opened in TTCN Suite.

For a full supported EBNF, see "The TTCN-MP Syntax Productions in BNF" on page 1574 in chapter 38, *Languages Supported in the TTCN Suite*.

The Standard MP Format

When you convert to MP, the TTCN document does not have to be analyzed or correct. TTCN-MP will be generated even for documents that are neither complete nor error-free. This implies that the generated MP file may be, but is not necessarily, conformant to the standardized TTCN-MP format

However, to ensure that the TTCN-MP document can be read by a non-TTCN Suite tool, you should analyze the document and correct any errors.

The IBM Rational MP Format

An optional but non-standard TTCN-MP for compact tables and ASN.1 references is supported. Additional fields in dynamic tables (fields which are transferred to test suite overview tables) are also supported. You are recommended to use this format for transferring TTCN document between the TTCN Suite instances and for making backups of the TTCN documents. Otherwise it works just as the standard MP format.

MP File Format Problem when Opening

When you open certain TTCN-MP documents, a problem with transferring the information (e.g. description) in the overview tables to the tables in Dynamic Part, may occur.

The TTCN standard allow path specifications to optionally include the document name first. This has the unfortunate effect that if the TTCN document contains a top level group with the same name as the TTCN document, there is, in general, no way of knowing if the first part of the paths is a group identifier or the TTCN document identifier.

TTCN Suite assumes that if the first part of the path is equal to the document name, it is the optional document name and, when converting, strips it away. When the document is saved as or exported to MP, the document name is always added to the front of all paths. That way, TTCN Suite is always able to open the MP files it exports.

If the TTCN document contains top level group identifiers equal to the document name, and TTCN Suite is unable to resolve the paths, temporarily change the document name in the MP file and change it back once inside TTCN Suite.

Note that ITEX 2.0 did not add the document name at export, and therefore the problem described may apply when open MP files exported by ITEX 2.0.

Fields Containing the '$' Character

The contents of table fields in a TTCN-MP file are usually ignored. However, to make it possible to open non-bounded free text fields (for example TS_VarValue field) with embedded dollar characters ('$'), those fields will be syntax checked. This means that it is not possible to convert TTCN documents containing unmatched single or double quotes in non-bounded freetext fields, that is, when you open a TTCN-MP file, syntax errors in those fields will not be tolerated.

Revision Control

There is no integrated revision control system in the TTCN Suite. Since normal visible files are used to store the TTCN documents, it is easy to integrate the TTCN Suite in a revision control system. However, since the `.itex` file format is binary, it is better to use the TTCN-MP format.

Converting to HTML

It is possible to export a TTCN document, or only a part of it, to HTML. The result will be one HTML-file containing what you selected to export, and – when applicable – hypertext links as well as information about the structure of the test suite will also be generated.

To convert a TTCN document to HTML:

1. Select the top node in the Browser.
   – You may also select only a part of the test suite, but multiple-selection is not possible.
2. Select *Save As* from the *File* menu.

   A dialog will be opened where you can save the document as HTML.

It is also possible to convert a currently opened table to HTML. To do this:

- Select *Generate HTML* from the pop-up menu in the Table Editor.

Crash Recovery

In the event of a TTCN Suite crash, it is possible to recover up-to-the-minute changes. For a description of the “iconcat” tool see “Crash Recovery” on page 1162 in chapter 24, *The TTCN Browser (on UNIX)*.

Shortcuts

Common Shortcuts

Shortcut	Action
Ctrl+N	Creates a new test suite.
Ctrl+O	Opens an existing test suite.
Ctrl+S	Saves the test suite to file.
Ctrl+P	Prints the test suite.
Ctrl+Z	Undoes latest editing action(s).
Ctrl+Y	Redoes latest undone editing action(s).
Ctrl+L	Shows/Hides Log Manager.
Ctrl+F	Opens a new Finder.
F7	Analyzes the selected node and its sub-tree.
F8	Generates code for the current test suite.
Alt+F8	Displays an options dialog to allow the user to set options for tools working on the Browser.
F5	Runs/continues current simulation.
Pause	Pauses current simulation.
Ctrl+Break	Aborts current simulation.
F11	Steps current simulation.

Browser Shortcuts

Shortcut	Action
<Up arrow>	Sets the focus to the above node in the Browser.
<Down arrow>	Sets the focus to the below node in the Browser.
<Right arrow>	If the selected node is collapsed and has children, it will be expanded. If the selected node is expanded and has children the selection will move to the first child.

Shortcut	Action
<Left arrow>	If the selected node is the root of an expanded sub-tree, the sub-tree will be collapsed. If the selected node is the root of a collapsed sub-tree the selection will move to the parent of the current node (if any).
<Ctrl+X>	Cuts the selected non-static node in the Browser.
<Ctrl+C>	Copies the selected non-static node in the Browser.
<Ctrl+V>	Pastes the contents of the cut/copy buffer after the selected node.
<Ctrl+Shift+V>	Pastes the contents of the cut/copy buffer as a child to the selected node.
<Delete>	Deletes selected node(s).
<Enter>	Opens a Table Editor for each selected node.
<Ctrl+A>	Adds a new table after the selected node.
<Ctrl+Shift+A>	Adds a new table as a child to the selected node.
Click	Sets selection on a node in the Browser.
Right-click	If the mouse pointer is above a node, sets selection on that node and display node specific pop-up menu. If no node is selected, displays background pop-up menu.
Double-click	If the selected node is a static node, the sub-tree will either be collapsed or expanded. If the selected node is a collapsed group, its sub-tree will be expanded. Otherwise the associated group table is opened. Finally, if the node is a non-static table a Table Editor for that node will be opened.
<Shift>-click	Selects/deselects all nodes between this selected node and the previously selected nodes.
<Ctrl>-click	Selects/deselects a node without affecting already selected nodes.

Table Editor Shortcuts

Shortcut	Action
<Tab>	Moves to the next editable field.
<Shift+Tab>	Moves to the previous editable field.
<Ctrl+Up arrow>	Moves to the previous line.
<Ctrl+Left arrow>	Moves to the next editable field.
<Ctrl+Down arrow>	Moves to the next line.
<Ctrl+Right arrow>	Moves to the previous editable field.
<Ctrl+Space>	Toggles selection of current row in a table body.
<Alt+Left arrow>	Moves forward in history.
<Alt+Right arrow>	Move back in history.
<Page Up>/<Page Down>	Scroll stable one page.
<F7>	Analyzes the current table.
<Ctrl+X>	Cuts selected rows.
<Ctrl+V>	Pastes rows after the selected row, or last if no selection exist.
<Ctrl+C>	Copies selected rows.
<F9>	Toggles breakpoint on selected row(s)
Right-click	Context sensitive popup.
<Ctrl>-right-click	Displays the definition of the currently selected identifier.
Click	Sets text selection and clear row selection.
<Ctrl>-click	Toggles row selection.

Log Manager Shortcuts

Shortcut	Action
<Ctrl+Alt+L>	Increases font size.
<Ctrl+Shift+L>	Decreases font size.

Finder Shortcuts

Shortcut	Action
<Up arrow>	Sets the focus to the above node in the *Results List*.
<Down arrow>	Sets the focus to the below node in the *Results List*.
<Ctrl+C>	Copies the selected non-static node in the *Results List*.
<Delete>	Deletes selected node(s).
<Enter>	Opens a Table Editor for each selected node.
Click	Sets selection on a node in the *Results List*.
Right-click	If the mouse pointer is above a node, sets selection on that node and display node specific pop-up menu. If no node is selected, displays background pop-up menu.
Double-click	Opens a Table Editor for the selected node.
<Shift>-click	Selects/deselects all nodes between this selected node and the previously selected nodes.
<Ctrl>-click	Selects/deselects a node without affecting already selected nodes.

Chapter 30 Editing TTCN Documents (in Windows)

Chapter

31

Analyzing TTCN Documents (in Windows)

This chapter contains a reference manual to the Analyzer in the TTCN Suite. You can also find information about how to find erroneous tables displayed in the Analyzer log.

The TTCN Browser and TTCN Table Editor are described in chapter 30, *Editing TTCN Documents (in Windows)*.

See chapter 2, *Introduction to the TTCN Suite (in Windows), in the TTCN Suite 6.2 Getting Started* for an overview of the TTCN Suite tool-set.

> **Note: Windows version**
>
> **This is the Windows version of the chapter. The UNIX version is chapter 26, *Analyzing TTCN Documents (on UNIX)*.**

Analyzing TTCN Documents

The Analyzer in the TTCN Suite does a complete syntax check on both TTCN and ASN.1 (as it is used in TTCN). The Analyzer also performs a number of static semantic checks, mainly the uniqueness and existence of identifiers.

For implementation reasons, the Analyzer does not exactly follow the TTCN standard – some syntax restrictions and relaxations apply. These differences will not affect the vast majority of users, but for reference they are documented together with the standard syntax in chapter 38, *Languages Supported in the TTCN Suite*. This chapter also includes the static semantics supported by the TTCN Suite.

During analysis, the TTCN Suite compiles an extensive symbol table containing all named objects in the TTCN document and the references between them.

Using the Analyzer

In the TTCN Suite, the Analyzer works on the currently active TTCN document. It is also possible to analyze a TTCN document from the Organizer, see "Analyze TTCN" on page 118 in chapter 2, *The Organizer*.

- To analyze a subtree of a TTCN document in the TTCN Suite, select a node in the view and then select *Analyze* from the *Build* menu. The selected node and all nodes in its subtree will be analyzed. If the root node is selected, the entire suite will be analyzed.
  – The shortcut is `<F7>`.

- To analyze the entire test suite, regardless of any selections in the test suite, select *Analyze Suite* from the *Build* menu.
  – The shortcut is `<Ctrl+F7>`.

- To analyze an entire modular TTCN system – containing the currently active suite or module – starting from the root suite, select *Analyze System* from the *Build* menu. This is equivalent to analyzing the root node in the Organizer.

  Note that it may take a while before this operation is completed, depending on the size and complexity of the system.

The Analyzer runs in background. The Progress Bar at the bottom of the TTCN Suite window shows the analysis completion. During the analysis it is possible to browse the already opened TTCN documents. It is not allowed to edit, save, close or open any of the TTCN documents during analysis. The analysis may be canceled by pressing the break button on the toolbar or by the menu choice *Build* > *Break*.

If an item is analyzed incorrectly, this will be recorded as an error message in the log. Items that have been found to be incorrect will be marked in red in all views of the document.

The *Analyzer* Dialog

When you analyze a document for the first time, the *Analyzer* dialog will be opened and you may change the options. Once this has been done, the same options will be used and the dialog will not be opened the next time you start the analysis from a quick-button or a keyboard shortcut. However, the dialog will always be opened when you select an analyzer menu choice. You can also open the dialog by the menu choice *Build* > *Options* or by the shortcut `<Alt+F8>`.

Note that every active TTCN document has its own analysis settings, so in practice, two or more TTCN documents can be edited and/or viewed at the same time in the TTCN Suite, and yet have individual settings. But multiple views of the same TTCN document do of course share the same settings.

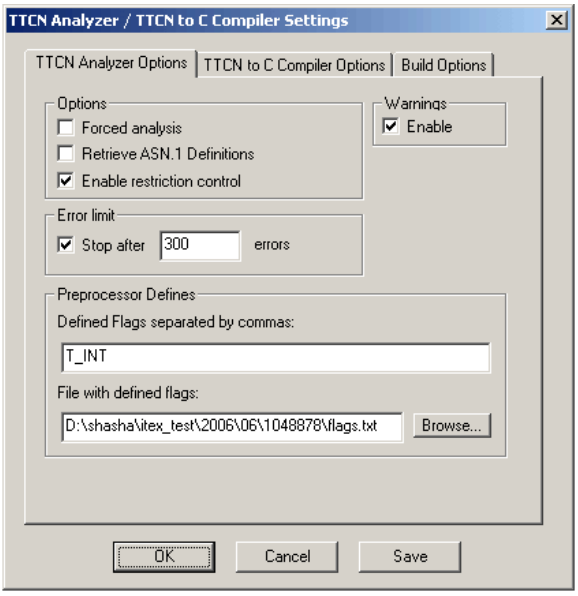


Figure 229: Analyzer dialog

The Analyzer reports all analysis results to the Log Manager. For each different test suite, a new tab will be created to allow the user to easily access the analysis information for different TTCN documents.

Note:

The output of the analysis will not be available if the Log Manager is closed and the option *Auto-Raise Log* is unchecked.

Forced Analysis

Indicates that all selected items are going to be analyzed and no item will be skipped. An item is normally skipped if it is OK.

Retrieve ASN.1 Definitions

Indicates that the definitions of encountered ASN.1 references should be retrieved. An ASN.1 definition is retrieved if this option is set or if the definition has not been fetched before.

Enable Restriction Control

Enables a set of extra controls done on the TTCN code to find even more semantic errors in the analyzed item.

Warnings

Gives more information.

Stop After n Errors

Stops the analysis when the number of errors is reached.

Defined flags

The flags defined for the TTCN Suite preprocessor are separated by commas (see "TTCN Suite Preprocessor" on page 1312).

File with defined flags

This field can be used to point to a file with flags defined for the TTCN Suite preprocessor. Use comma (",") to separate the flags in this file.

Error Messages

Errors detected during analysis are recorded in the log. For the fields in the header of a table, the error messages have the general format:

```
================================
Table name and table type
Field type 1
Error message
Field type 2
Error message
:
:
Field type n
Error message
================================
```

Example 217 ————————————————————————————————

An error found in the default name (identifier) field of the default dynamic behaviour called UT_DEFAULT:

```
========================================
Analysis messages in table: UT_DEFAULT of type:
Default Dynamic Behaviour
 Default Name:
```

```
   The referenced identifier UPPER_PCO is not declared.
   UT_DEFAULT (p : UPPER_PCO)
   ----------------^
   ==========================================
```

In the body of a table (where the field type name does not uniquely define a field) the row number is also included:

- Row number
- Body type
- Error message

Example 218

Errors found in a test step table header (the default field) and in the body (rows 1 and 2 of the behaviour description):

```
   ==========================================
   Analysis messages in table: ESTABLISH_CONNECTION of
   type: Test Step Dynamic Behaviour
    Default:
   Mismatched number of parameters: is 1, should be 0.
      Row: (#1)
         Behaviour Description:
   The referenced identifier CR is of wrong type.
   UPPER_PCO ! CR
   ------------^
      Row: (#2)
         Behaviour Description:
   The referenced identifier CC is of wrong type.
   UPPER_PCO ? CC
   ------------^
   ==========================================
```

Resolving Forward References

During analysis, the TTCN Suite does two main tasks:

- It checks the **syntax** of the abstract TTCN document.
- It checks some of the **static semantics** of the TTCN document.

The syntax of TTCN is defined in ISO/IEC 9646-3 Annex A as a list of BNF productions or rules. See also "The TTCN-MP Syntax Productions in BNF" on page 1574 in chapter 38, *Languages Supported in the TTCN Suite*. The TTCN Suite checks all these rules except for a very few minor deviations; for more information, see "The Restrictions in the TTCN Suite" on page 1570 in chapter 38, *Languages Supported in the TTCN Suite*. If a rule is not followed in the TTCN document then this will be reported as an error by the Analyzer.

The static semantic rules of TTCN are also defined in Annex A of ISO/IEC 9646. These are in the form of text statements which *limit* the use of some of the syntax rules, usually in a specific context. "TTCN Static Semantics" on page 1603 and "ASN.1 Static Semantics" on page 1619 in chapter 38, *Languages Supported in the TTCN Suite* describe the static semantics which are controlled by the Analyzer. The static semantics currently supported are mainly of the following types:

- Identifiers (names) are checked for uniqueness with respect to the scoping rules defined in ISO/IEC 9646-3
- The existence of referenced TTCN objects (e.g. a test step) from another TTCN object (e.g. test case) is verified
- Type control and type restriction control

In order to achieve this, the Analyzer constructs a symbol table of all the named TTCN objects that it finds during analysis, e.g. variable names, type identifiers, tables, etc.

It is important to note that TTCN allows *forward reference*s. For example a test step may attach another test step that is declared **later** on in the TTCN document, that is, the referenced (i.e. attached) test step, appears **after** the test step that has done the attach.

The Analyzer has a feature called *back-pass* designed to resolve forward references. The back-pass is automatically invoked at certain points in the analysis process.

Forward references usually appear in the context of ASP/PDU definitions, constraints and as the result of attachment statements in behaviour trees. The reason for this is that the Analyzer does not traverse the selected items in exactly the listed order. The Analyzer order deviates from the listed order in three places:

- Items in the PDU type definitions sub-tree are analyzed before items in the ASP type definitions sub-tree.
- Items in the PDU constraint declarations sub-tree are analyzed before items in the ASP constraint declarations sub-tree.
- Items in the defaults library sub-tree are analyzed before items in the test step library sub-tree and those are analyzed before items in the test cases sub-tree.

ASN.1 External Type/Value References

References to ASN.1 definitions in external ASN.1 modules can occur in the following four TTCN tables:

- Test suite constant declarations by reference
- ASN.1 type definitions by reference
- ASN.1 ASP type definitions by reference
- ASN.1 PDU type definitions by reference

In these tables there are three important fields to consider:

- *Module Identifier* (set by user)
  The name of the ASN.1 module. This should be the real name of the model, i.e. the name stated inside the module itself.
- *Type Reference/Value Reference* (set by user)
  This is the name of the desired type (or value).
- *Type Definition/Value* (set automatically when fetched)
  This is the field where the fetched type definition/value is copied into.

The term *definition* is used below both to denote an ASN.1 type definition and an ASN.1 value.

When an ASN.1 reference is analyzed, the referred definition must be available to the Analyzer. In the pro formas for this kind of reference in the TTCN Suite, there is an external column where both the definition and the parse tree is stored.

If the Analyzer encounters any reference to a type/value, the definition is fetched, given the type/value name and the module identifier, and copied into the external field before the field is analyzed. The operation of fetching the definition of the reference is controlled by an option in the Analyzer dialog. Important to note is that the fields *Module Identifier* and *Type/Value Reference* are not parsed. This means that if a change is made in either of these two fields, the row will not be analyzed and thus no fetching will be made. The solution to this is to set the *Enable Forced Analysis* in the Analyzer dialog when altering a reference. See "The Analyzer Dialog" on page 1303.

Analyzing ASN.1 References

This is the general algorithm when encountering an ASN.1 in the analysis phase.

1. First of all, an attempted fetch is only made if no current definition exists or the Analyzer fetching-option is selected.
2. The actual fetching is made, see "Retrieving external ASN.1 Definitions" on page 1309.
3. If the definition is successfully fetched, the identifiers in the definition is converted to TTCN compatible syntax, see "Syntactic Conversion" on page 1310.
4. Finally a possible update of the definition is made and the analysis is continued.

Retrieving external ASN.1 Definitions

From the ASN.1 module a simple parse tree is built to access its contents. The parse tree is (re-)built when either it has not been built before or the ASN.1 module has been modified since last access. The parse tree for an ASN.1 module "lives" for the duration of the whole analysis phase.

Syntactic Conversion

The received definition must be manipulated in the following sense. All identifiers must be syntactically checked and altered if they include any characters that is disallowed in TTCN. The rule is that all '-' (dash) characters in an ASN.1 name are replaced by a '_' (underscore).

No special care has to be considered to ASN.1 comments since they are ignored when the parse tree is built from the module.

External type references in the type definition (e.g "... Module.Type...") will only be converted like above and left for the Analyzer to deal with (see also "Restrictions" on page 1310).

Restrictions

The following features are not supported in TTCN:

- External type reference identifiers on the form *ModuleIdentifier.Type/Valuereference* are not supported.
- Type/value references within the same ASN.1 module are not supported.

Furthermore, there are restrictions in the ASN Utilities when it is used to extract the external types and values. These restrictions are introduced in translating ASN.1 to SDL but since the same algorithm is used for extracting types and values, the restrictions are also relevant for the TTCN Suite (see "Translation of ASN.1 to SDL" on page 704 in chapter 13, *The ASN.1 Utilities*).

Error Handling

Any error that occurs while finding, parsing or accessing the ASN.1 module will cause the old definition to remain. Only when a completely successful fetching has been made, the definition will be updated. Possible errors could be:

- The ASN.1 module not found among the dependencies.
- The ASN.1 module file could not be found or was not readable.
- The ASN.1 module file was not syntactically correct.
- The referred type/value could not be found in the module.

See chapter 13, *The ASN.1 Utilities* for a more general view on ASN.1 usage.

TTCN Static Type Restriction Control

The TTCN standard defines a set of semantic rules for type definitions. In particular it defines simple types as types which might contain a true subset of the values which the parent type might contain. The Analyzer now analyzes the TTCN type system and also reports violations of these rules. This analysis is implemented as a post-pass over the related tables, and is only applied when the previous passes have been successful.

Furthermore, restriction control is performed on a number of other constructs in the TTCN language, thus facilitating the task of writing semantically correct TTCN documents. Most of the TTCN types and values are checked and thereby reducing the chance of programming errors slipping through to other tools which depends upon the correctness of the TTCN document.

Application of Static Type Restriction Control

Static type restriction control is applied to at least the following TTCN tables and fields:

- Simple type definitions.
- TTCN structured type / ASP / PDU / CM definitions.
- Test suite constant declarations.
- Test suite variable declarations.
- TTCN structured type / ASP / PDU / CM constraint declarations.
- Actual constraint reference parameter lists.
- Actual parameter lists.

The static type restriction control also evaluates expressions in advance, where possible, and checks that the result does not violate the type restrictions of the field where the value is found. It operates on both types and values, regarding specific values as special cases of types.

TTCN Suite Preprocessor

The TTCN Suite preprocessor is a text processor that manipulates the text of a TTCN table as part of the first phase of Analyzing. TTCN Suite preprocessor allows you to produce different ETS code depending on the flags set. The syntax of TTCN Suite preprocessor directives is similar to directives for a C preprocessor:

```
#if <expr>
....
[#else]
...
#endif

<expr> ::= defined(<flag>) | '!' <expr> | <expr>
'||' <expr> | <expr> '&&' <expr> | (expr)
```

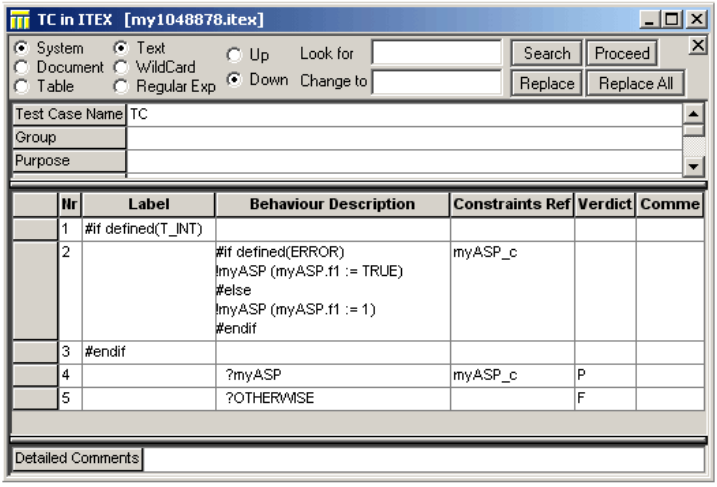


Figure 230: Example of preprocessor directives

These directives can be used in every editable field, not different from TTCN and ASN.1 definitions. The '`#if`' directive and its corresponding '`#endif`' directive must be placed within the same field, except when they are used in the body of a TTCN table that may consist of several rows. In this case it is allowed to use directives '`#if`', '`#else`' and '`#endif`' in separate rows, and only one such directive in the first field is allowed in such a row. The scope of such a directive will be all rows between.

The defined flags for Analysis and Code Generation are set in the Analyzer options (see "The Analyzer Dialog" on page 1303) The flags can also be set via options '-i' and '-I' from a command line session (see "Running the TTCN to C Compiler from the Command Line" on page 1323 in chapter 32, *The TTCN to C Compiler (in Windows)*).

Finding Tables from the Analyzer Log

When you are debugging a TTCN document, it may be useful to search for and display named tables. The search will include the entire current TTCN document – no selections are required.

To find an erroneous table identified in the Analyzer log:

1. Place the cursor on the name of the table in the log and then <Ctrl>-right-click the name of the table.
2. In the popup menu that will be opened, select the table name (at the top of the menu), and the table will be opened.

You can also use the Finder to find a table. See "Finding and Sorting Tables" on page 1282 in chapter 30, *Editing TTCN Documents (in Windows)*.

Chapter

32

The TTCN to C Compiler (in Windows)

This chapter describes what the TTCN to C compiler is used for, how to run it and the structure of the generated code.

When the TTCN to C compiler has translated TTCN into C code, the code must be adapted with the system that is to be tested. The adaption process is described in chapter 36, *Adaptation of Generated Code*.

Note: ASN.1 support

The TTCN to C compiler supports only a limited subset of ASN.1. See "Support for External ASN.1 in the TTCN Suite" on page 733 in chapter 13, *The ASN.1 Utilities, in the User´s Manual* for further details on the restrictions that apply.

Note: Windows version

This is the Windows version of the chapter. The UNIX version is chapter 27, *The TTCN to C Compiler (on UNIX)*.

Introduction to the TTCN to C Compiler

When developing new systems or implementations of any kind, the developing process is divided into well defined phases. Most commonly, the first phases involve some kind of specification and abstract design of the new system. After a while the implementation phase is entered and finally, when all parts are joined together, the test phase is activated.

In any case when a system is tested, we want to make sure that its behavior conforms to a set of well defined rules. TTCN was developed for the specification of test sequences. Unfortunately, as very few systems interpret or compile pure TTCN, we need to translate the TTCN notation into a language which can be compiled and executed. In the case of the TTCN Suite, the TTCN to C compiler translates TTCN to ANSI-C.

Even after the TTCN code has been translated, there are a couple of things that need to be taken care of. In this case, we must *adapt* the generated code with the system it intends to test. This chapter describes how the TTCN to C compiler is used. The adaption process is described in "Adaptation of Generated Code" on page 1487 in chapter 36, *Adaptation of Generated Code*.

Getting Started

Unfortunately, a test sequence description expressed in TTCN cannot easily be executed as it is. This, because the test notation is not executable and only few test environments interpret pure TTCN. A different approach to create an executable test suite (ETS), is to translate the formal test description into a language which can be compiled into an executable format.

The TTCN to C compiler translates TTCN into ANSI-C which can be compiled by an ANSI-C compiler. Figure 231 depicts the first step in the process of creating an ETS using the TTCN to C compiler.

The generated code, called the TTCN runtime behavior, is only one of the two major modules of an ETS.

The second module which is needed includes test support functions which are dependent on the protocol used, the host machine, test equipment, etc. For this reason, it is up to the user to write this second module and in such way adapt the TTCN runtime behavior to the system he/she wants to test.

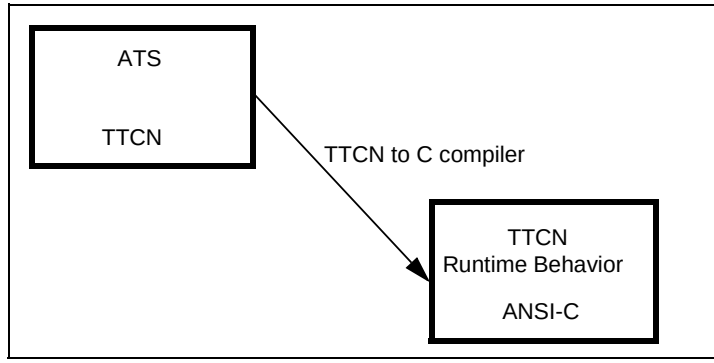


Figure 231: Translating TTCN to ANSI-C

The adaption process is described in "Adaptation of Generated Code" on page 1487 in chapter 36, *Adaptation of Generated Code*. Figure 232 displays the anatomy of the final result.

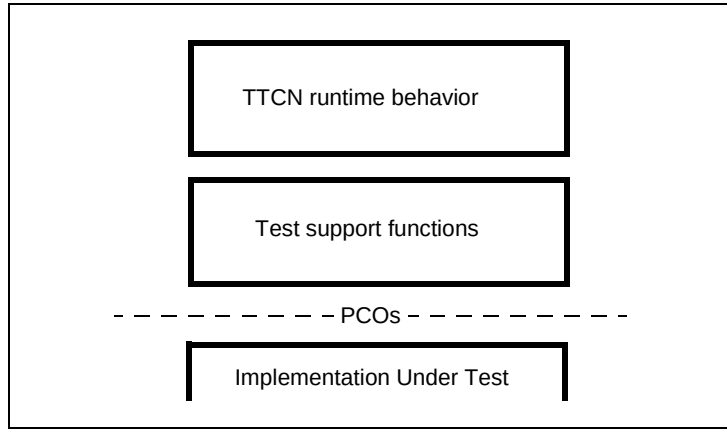


Figure 232: The anatomy of an ETS

Running the TTCN to C Compiler

To generate C code for the currently selected test document:

- Select *Generate Code* from the *Build* menu.
  – The shortcut is `<F8>`.

  This will open a dialog where you may change settings for code generation (see Figure 233).

  You can also open this dialog by the menu choice *Build* > *Options* or by the shortcut `<Alt+F8>`.

> **Note:**
>
> Observe that the TTCN to C compiler is unable to function correctly if the test suite is not fully verified correct. This verification is accomplished by running the Analyzer tool on all the parts of the test suite.

Note that every active TTCN document has its own code generation settings, so in practice, two or more TTCN documents can be edited and/or viewed at the same time in the TTCN Suite, and yet have individual settings. (Multiple views on the same TTCN document does of course share the same settings.)

The TTCN to C compiler runs in background. The Progress Bar at the bottom of the TTCN Suite window shows the TTCN to C compiler completion. During the code generation it is possible browse the already opened TTCN documents. It is not allowed to edit, save, close or open any of the TTCN documents during analysis. The analysis may be canceled by pressing the break button on the toolbar or by the menu choice *Build* > *Break*.

Options

- If the *Ignore bodies of test suite operations* option is checked, the compiler will ignore to generate code for test suite operations. This is useful if you already have a file with your own test suite operations that only need to be linked to the rest of the code. If the option is not checked, the test suite operators will be generated and included in the file `tsop_gen.c`.

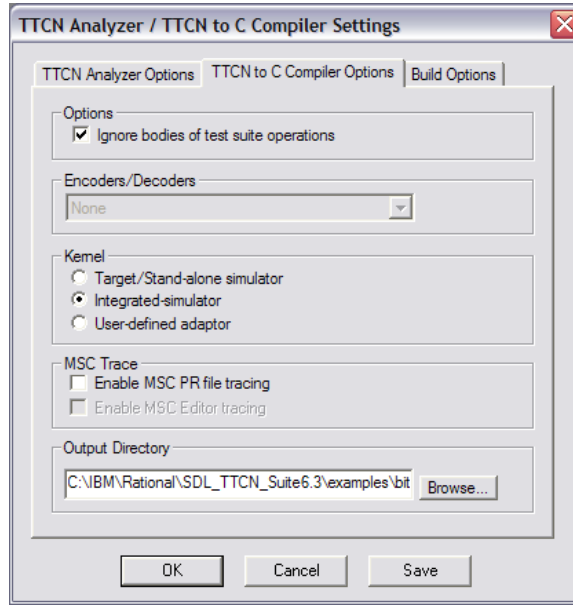


Figure 233: TTCN to C Compiler Options tab

Encoders/Decoders

- By selecting one of the alternatives *None*, *Generate BER encoders/decoders* or *Generate PER encoders/decoders* in the drop down list the compiler will either omit generation of encoders/decoders or generate extra code for BER or PER encoding and decoding support, see chapter 36, *Adaptation of Generated Code*.

Kernel

- To build a TTCN Exerciser, select the option *Target/Stand-alone simulator*. For more information about the TTCN Exerciser, see chapter 34, *The TTCN Exerciser*.
- Select *Integrated-simulator* if you want to execute a test suite together with a simulated SDL system.
- To build a user-defined adaptor, select *User-defined adaptor*.

MSC Trace

- *Enable MSC/PR file tracing*

  Generates MSC/PR format files that are saved in the current working directory. By default, the MSC file names will be on the form `log_<TestCaseId>_<SequenceNo>.mpr`, where `<TestCaseId>` is substituted by the test case name of the logged test case. `<SequenceNo>` is an integer that is started by 0000 and increased by one if there is already a version n in the working directory.

  The preprocessor constant MSC_FILE_MODE should be defined at compile-time of `mscgen.c` to get this mode. The constant GENMSC should be defined for compiling `globalvar.c` to activate appropriate calls.

  If the file cannot be created, a new attempt will be done at the start of the next test case. No file log will be created.

  The function `MscSetPrefix` can be used to change the path and prefix of generated files at runtime.

- *Enable MSC Editor tracing*

  Creates a new diagram in the MSC Editor when a new test case is started and then appends and displays the events as they are executed. This mode assumes that an MSC Editor license is available and that TTCN Suite is running at the host where the ETS is running and provides run-time MSC logging.

  The preprocessor constant MSC_MSCE_MODE should be defined at compile-time of `genmsc.c` to get this mode. The constant GENMSC should be defined for compiling `globalvar.c` for activation of the appropriate calls.

  If the creation of events in the MSC Editor fails, it will be retried at the next event, possibly creating an inconsistent MSC.

  This mode may also require access to the Public Interface libraries and include files that can be found in the installation.

For more information, see "TTCN Test Logs in MSC Format" on page 1328.

Output Directory

The final thing you have to do is to set the output directory.

OK

When you have set the options, click OK to start the generation of code. When the generation phase is started, information about the code generation will be displayed in the Log Manager. You will receive information about what parts that were generated and also some statistics about the amount of tables traversed during the generation phase.

If code is being generated for a large test suite, the status bar will show the progress.

Save

If you would like to store the option settings for future usage, press the *Save* button. The options settings is then saved in a file with the same name as the current test suite's filename (with extension) and with an additional '.ini' extension. For example will the options for 'foo.itex' be saved in a file named 'foo.itex.ini', for a test suite named 'foo.mp' the options will be in the file 'foo.mp.ini'. The options file is placed in the same folder as the test suite file.

The options file is automatically searched and the options settings are loaded when a TTCN Suite browser window is opened (on Windows), or when the TTCN Suite Make dialog is opened (on UNIX).

Build Options

Makefile Type

Choose one of the known make types, corresponding to your compiler, for the generated makefile.

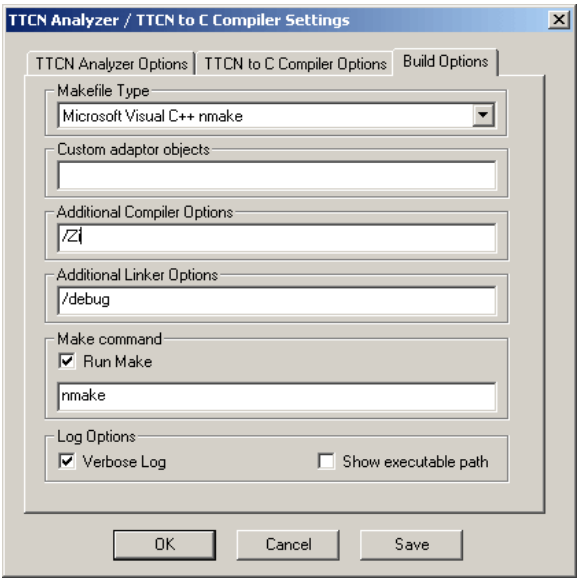


Figure 234: Build Options tab

Custom adaptor objects

This text field allows you to specify additional object files to be linked to ETS.

Make command

The Run Make check box controls if the make command is to be invoked directly after the code generation (provided it has completed successfully). The text field allows a manual change of the invoked make command if this is desired.

Log Options

There are two options in the Build Options tab for log management. These options are also related to a compilation flag, `SHOWSIGNALPARAMETERS`, which is generated in the makefile.

Verbose Log

The *Verbose Log* switches the logging between the default mode, which prints the result of the Make process as it goes along, and the *Verbose* mode, where all possible information is presented. The *Verbose* log is primarily intended for debugging.

The *Verbose* log contains the following information in addition to the default log:

- signal parameters, with an attempt to restore all names (for signal, fields, parameters, choices, etc.), even when these names are lost (which is the case when the user construct the signals directly in the adaptor). The default log can be made to show this if the Compilation flag /DSHOWSIGNALPARAMETERS is set.
- matching processes are presented *Verbose*. Each field shows how it was matched, using the indentation level for nested fields.

Show Executable Path

When *Show Executable Path* is selected, the lines in the ETS log will be prefixed by the path for the executing Test Case and Test Steps, like the following example:

```
Test Case:[line number]  Step_no:[line number] ... Test Step_n
```

Compilation flag /DSHOWSIGNALPARAMETERS

The flag `SHOWSIGNALPARAMETERS` is generated in the Makefile even when the log is not *Verbose*. With this compilation flag the default log will also contain the complete contents of all signals, just like when log is set to *Verbose*. When this is not desired, this flag can easily be removed from the Makefile.

Additional Compiler Options and Additional Linker Options

Here it is possible to specify any compiler and linker options that will be added into the generated Makefile.

Running the TTCN to C Compiler from the Command Line

You can start the TTCN to C compiler by executing `ttcn2c` with command line switches. The usage is `ttcn2c [switches] filename`.

Example 219 ----------

To generate C code for `example.itex` in the current directory, with hard values and in silent mode, use the command:

```
ttcn2c -ms example.itex
```

C Code Generator Parameters

Switch	Explanation
`-a <objs>`	Link with `objs` as adaptor
`-A <ASN1files>`	File containing filenames of ASN.1 module files.
`-b <dir>`	Search in `<dir>` for static files
`-B`	Generate BER encoding/decoding interface
`-C <options>`	Set the additional compiler options
`-d`	Debug information in C code
`-D`	Step through ETS execution
`-e <file>`	Use `file` to be included in the make file
`-f`	Forced mode – always overwrite files
`-F`	Definitions of encountered ASN.1 references should be retrieved. See "Retrieve ASN.1 Definitions" on page 1193 in chapter 26, *Analyzing TTCN Documents (on UNIX)*
`-g`	Ignore bodies of test suite operations See also "Options" on page 1318.
`-h`	Help
`-i <defined flag>`	Set the flag defined for the preprocessor
`-I <flags file>`	Set the file with the flags defined for the preprocessor.
`-j <file>`	User file name for types
`-k <file>`	Use file name for constraints

Switch	Explanation
`-l <file>`	Specify log file
`-L <options>`	Set the additional linker options
`-M file`	Enable MSC/PR file tracing
`-M msce`	Enable MSC Editor tracing
`-o <dir>`	Specify output directory
`-p <file>`	Use file name for behavior
`-P`	Generate PER encoding/decoding interface
`-r <file>`	Use file name to generate a makefile
`-R`	Disable makefile generation
`-s`	Silent mode – do not display messages
`-t`	Generate simulator See also "Kernel" on page 1319.
`-T`	Generate targe/stand-alone kernel
`-u vc`	Generate code for Microsoft Visual compiler
`-w`	No warning mode
`-z <lines>`	Aim at splitting generated file at <lines> number of lines

Generating C Code for Modular Test Suites

Generating C code for modular test suites from the command line, may be a bit tricky. Please see the example below for an explanation of how to proceed.

Example 220 ————————————————————————————

You want to generate C code for the modular test suite `mts.itex`, which depends on `module1.itex`, `module2.itex`, `module3.itex` and `module4.itex`. To do this enter:

```
ttcn2c """mts.itex""""""module1.itex""""""module2.itex""""""module3.itex"
""""""module4.itex"""
```

Note that there are six quotation marks between the file names.

This will generate code for the entire modular test suite.

It is also possible to use MP documents in the same manner.

For a description of file naming, see "Files in the TTCN Suite" on page 24 in chapter 2, *Introduction to the TTCN Suite (in Windows), in the TTCN Suite 6.2 Getting Started*.

What Is Generated?

Figure 231 shows how ANSI-C code is generated from a TTCN test suite. The generated code alone is not executable as it needs the test support functions module (see Figure 232).

In the case of the code generated from the compiler we also need a set of static functions which handle TTCN basics and other internal events. Even if these functions are vital for the successful compilation and execution of the generated code, the user should not have to worry about this part. These functions are gathered in a small set of static files which are compiled by the generated makefile and linked with the rest of the code.

The Code Files

The generated makefile is the file containing a definition of how the code should be compiled and linked. This will not be needed if you are compiling the generated code in a separate development environment.

The `adaptor.h` and `adaptor.c` files are the files that contain the adaptation code. If code is generated for the first time, these files will be generated by the compiler with empty function templates for the user to implement. On the other hand, if these files are present in the target directory the user does not have to worry about getting them overwritten.

The `*_gen.{c,h}` files contains the generated code from the TTCN test suite.

The `asn1ende.h` file contains the encode and decode functions for the ASN.1 Types. See chapter 58, *ASN.1 Encoding and De-coding in the SDL Suite, in the User´s Manual*. (Only if ASN.1 support has been selected)

The Adaptation

We are now ready to deal in greater detail with the adaptation phase which is the final phase to create an executable test suite. The adaptation process is described in "Adaptation of Generated Code" on page 1487 in chapter 36, *Adaptation of Generated Code*.

TTCN Test Logs in MSC Format

The TTCN standard conformance log, though it is a relatively complete logging format, has a few drawbacks for some purposes:

- The conformance log is very detailed.
- The conformance log requires knowledge of TTCN and its semantics.
- The conformance log is not always particularly easy to read.

In order to get a more high-level view, the log must either be filtered or another approach needs to be taken – in this case the MSC log format. The current version of the TTCN Suite has support for an additional logging mode, to generate MSC/PR standard conformant logs – automatically saved in files and also directly to the MSC Editor.

For limitations in the MSC logging, see "MSC Logging" on page 41 in chapter 1, *Compatibility Notes, in the Release Guide*.

MSC Logging Applications

There are several applications of MSC logging. Here are some:

- Test reviews

  The MSC log format is very readable and can be used for reviewing test progress and for evaluating test results with requirements specifications.

- Test result recreation

  If an SDL model has been used to specify the system behavior, a composed MSC log can be used to re-create the test result by using the SDL Explorer's Verify-MSC feature. This will allow a particular test behavior to be handed to system designers who can recreate the result on a workstation level.

- Test re-engineering

  If MSC traces are being used to generate test cases, the tests can be modified at the MSC level. A modified MSC can be used to generate a new tests. For instance – a test written in TTCN generates an MSC. At a review of the MSC logs a new scenario may be identi-

fied. The MSC can be changed to reflect the new scenario, and the MSC can be translated back to TTCN.

MSC Logging Modes

There are three major modes of logging, which can be toggled between the test cases. It is not possible to change the logging mode in a running test case since that would cause an inconsistency in the MSC file. The modes are:

- None

  No MSC logs are generated at all. This mode improves performance somewhat compared to the other modes. For best performance, the MSC generation code can be entirely disabled. For more information see "Compiling an ETS with MSC Generation" on page 1330.

- Composed

  MSC generation of externally visible events of the ETS. This is suitable for test regeneration and for re-viewing test results. The performance is better than Decomposed mode. The generated MSC views the ETS as a set of PCOs that interact with a IUT. The ETS is not visible as one particular instance. Some relative timing information may be lost in this mode.

- Decomposed

  MSC generation of externally visible and internal events of the ETS. This is the most detailed mode and it produces MSC events for almost all actions of the ETS. The purpose of the MSCs that are generated is to visualize the internal events and configuration of the ETS, and for test case debugging purposes.

MSC Instances Generated

Entity	Composed	Decomposed	Comment
SUT	Yes	No	
PCO	Yes	Yes	
MTC	No	Yes	
PTC	No	Yes	Dynamic

Events Logged

Event	Composed	Decomposed	Comment
Final Verdict	Yes	Yes	MSC Text
Preliminary Verdict	No	Yes	MSC Condition
Create	No	Yes	
Done	No	Yes	
Send to PCO	Yes	Yes	
Send to CP	No	Yes	
Receive from PCO	Yes	Yes	
Receive from CP	No	Yes	
Implicit send	No	Yes	
Start Timer	No	Yes	
Cancel Timer	No	Yes	
Timeout Timer	No	Yes	
Implicit Cancel	No	Yes	End of TC
Implicit Receive	No	Yes	End of TC
Message Values	Yes	Yes	

Compiling an ETS with MSC Generation

The ETS Generator can be compiled for a workstation or an embedded environment. Some modifications need to be done for applying it in the embedded environment if it lacks a file system. By default the tool will allow for any of these combinations at code generation time. For more information see "MSC Generation Modifications" on page 1331.

Definitions that may be changed to customize the MSC generation:

Definitions	Explanation
MSC_DEFAULT_SYSTEM_NAME	Name of IUT instance in composed mode.

Definitions	Explanation
MSC_MAX_VALUE_LEN	Maximum length of an encoded message value.
MSC_NO_VALUE_LOGGING	If defined, no message values are displayed – this may make the ETS more efficient and the MSC logs more readable when using complex messages.

MSC Generation Modifications

Modifications to the MSC generation for running on an embedded environment include:

- The MscAppendPREvent function can be modified to transfer the PR events via any type of uni-directional communications device. It is recommended to create another definition similar to the MSC_MSCE_MODE or MSC_FILE_MODE and complement the `mscgen.c` file with these.
- Any place where MSC_FILE_MODE or MSC_MSCE_MODE is used should be having a new definition for the new logging mode.

Chapter 32 The TTCN to C Compiler (in Windows)

Chapter

33

The SDL and TTCN Integrated Simulator (W)

This chapter describes the SDL and TTCN Integrated Simulator in the Windows version of the TTCN Suite. It gives an introduction to the concept of the SDL and TTCN Integrated Simulator, as well as a guide to its functionality.

Note: Windows version

This is the Windows version of the chapter. The UNIX version is chapter 28, *The SDL and TTCN Integrated Simulator (U)*

The SDL and TTCN Integrated Simulator

The SDL and TTCN Integrated Simulator allows you to generate executable test suites (ETS) for testing of simulated SDL systems. This allows the testing of the system design early in the design process.

These ETSs allow you to execute test cases and/or test groups (hereafter referred to as tests) at full speed or by single stepping through the selected tests. There is also the possibility to set breakpoints in the tables of the tests.

It should be noted that there is no separate SDL and TTCN Integrated Simulator GUI. The SDL and TTCN Integrated Simulator will use the Table Editor to present the current line during execution.

More information about the SDL Simulator can be found in *chapter 49, The SDL Simulator*.

Performing an Integrated-Simulation with the SDL Suite

Given a test suite containing a set of tests you want to execute together with a simulated SDL system, the following steps are to be performed:

1. In the TTCN Suite, analyze the test suite and make sure it contains no errors.
2. In the TTCN Suite *TTCN to C Compiler Settings* dialog, generate an executable test suite.

   The following *TTCN to C Compiler Options* should be set:

   – *Integrated-simulator*

   – *Ignore bodies of test suite operations*

   In the *Build Options* tab:

   – You should set *Run Make*.

   – You may have to change the *Makefile Type*.

   For more information about the TTCN to C compiler and the options, see chapter 32, *The TTCN to C Compiler (in Windows)*.

3. Start the SDL and TTCN Integrated Simulator by selecting *Invoke Integrated-Simulatator* from the *Co-Simulate* menu.

   In the file dialog that will be issued, select the simulator ETS – that is, the executable produced in step 2. After that, the TTCN-SDL Integrated Simulator toolbar will appear.

4. Generate and start the SDL Simulator, if not already done. In the SDL Simulator UI, load the SDL Suite generated simulator executable.
5. In the SDL Simulator, give the command **`start-itex`**.
6. In the TTCN Browser, select the tests you wish to execute.
7. Click the *Run/Continue* or *Step* button, to start the actual integrated-simulatation in the TTCN Suite.
8. In the SDL Simulator, give the command **`go-forever`**.

The SDL and TTCN Integrated Simulator Operations

To abort the test in progress:

- Select *Co-Simulate* > *Abort*.
  The SDL Simulator may have to be restarted after this step. A new integrated-simulation can be started now.

To temporarily pause the test simulation:

- Select *Co-Simulate* > *Pause*.
  The integrated-simulation can be continued by doing a “run” or “step” operation.

To step one line of TTCN code:

- Select *Co-Simulate* > *Step*.

To run/continue a test at full speed until it is finished or until a breakpoint is reached:

- Select *Co-Simulate* > *Run/Continue*.

To toggle breakpoint status at currently selected lines in the Table Editor:

- Select *Co-Simulate* > *Toggle Breakpoint*.

Integrated-Simulation from command line

The generated ETS for Integrated-Simulation may be started from command line. Do this without any arguments in command line. In this case ETS has no any graphic interface. It reads commands from stdin and print the results of performed commands, Conformance Log, etc to stdout. Extra debug output will be printed if the environment variable TTCN_SIMULATOR_DEBUG is set.

See "Integrated-Simulation from command line" on page 1232 in chapter 28, *The SDL and TTCN Integrated Simulator (U)*.

Performing "batch mode" Integrated-Simulation

The generated ETS for Integrated-Simulation may be started together with SDL Simulator in "batch mode" (Windows only). It reads the commands from the specified file and executes them one by one. For such execution start the TTCN ETS for Integrated-Simulation and as an argument in the command line you pass the name of the file containing the commands. The SDT Postmaster must be started before the TTCN Integrated-Simulator (the Postmaster is running in the background of the SDL Suite Organizer).

Example 221 ––

```
TTCN_sim.exe simcommand.txt
```

––

The file with commands may contain any of the commands described in chapter above. It should contains 'run-SDL' command for starting the SDL simulator.

Example 222: Contents of “simcommand.txt”––––––––––––––––––

```
run-SDL sim_5918_smc.exe
SDL-command sta-it
SDL-command go
run test
quit
```

––

The Integrated-Simulator prints the result of the performed commands, Conformance Log, etc to stdout, it is also possible to redirect this output. The Final Verdict may be found in output between the `<VERDICT>` tags:

```
<VERDICT>
PASS
</VERDICT>
```

Information Messages

The SDL and TTCN Integrated Simulator can present messages to the user in three forms:

- Informative messages
- Warning messages
- Error messages

Informative Messages

Informative messages are normally presented at the status bar. These might be for instance progress reports or other simple messages, which are only of a temporary interest (such as the tool tips).

Warning Messages

Warnings, also known as “non-critical errors”, are events that indicate a problem during the preparation for a integrated-simulation, or during the actual integrated-simulation run. These are presented in standard warning dialogs.

Error Messages

Critical errors will halt the integrated-simulation. They are presented by an error dialog that appears in front of the application. The error dialog can often appear large and intimidating, but it simply gives a more verbose reason to why the error occurred, and what steps should be taken to avoid it.

Troubleshooting

The Simulators Stop Communicating

The SDL Simulator and the SDL and TTCN Integrated Simulator may stop communicating. This can occur if there are “old” instances of the SDL and TTCN Integrated Simulator ETS process still alive. They will then “steal” messages from the intended recipient. There are some ways to make the Simulators communicate again:

- Exit the SDL and TTCN Integrated Simulator and start it again.
- Exit the SDL and TTCN Integrated Simulator. Restart the SDL Simulator. Start the SDL and TTCN Integrated Simulator again.
- Exit the Simulators and the Organizer. Check that no Postmaster process and no SDL and TTCN Integrated Simulator ETS process remains. If they do, kill them. Start the Organizer, the SDL Simulator and the SDL and TTCN Integrated Simulator.

Test Execution Stops

The execution of tests may stop at a certain point during execution for no reason. If the problem is reproducible, you should check the supported types. There is a possibility that a type has been used in a message between the simulators that is not supported.

The Simulators Get Out of Sync

The simulators may get out of sync. One way of doing this is when an execution in the SDL and TTCN Integrated Simulator has been aborted and the SDL Simulator has not been restarted.

- Press the abort button in the SDL and TTCN Integrated Simulator. Restart the SDL Simulator.

Type Mappings in Integrated-Simulation

The data type mapping used by the integrated-simulation of the SDL simulator and the SDL and TTCN Integrated Simulator in Windows is the same as for the integrated-simulation on UNIX. See “Type Mappings in Integrated-Simulation” on page 1238 in chapter 28, *The SDL and TTCN Integrated Simulator (U)*.

Chapter

34 *The TTCN Exerciser*

A TTCN Exerciser is built from the TTCN to C compiler. For more information, see chapter 27, *The TTCN to C Compiler (on UNIX)* and chapter 32, *The TTCN to C Compiler (in Windows)*.

For information about TTCN Exerciser limitations, see "TTCN Exerciser" on page 44 in chapter 1, *Compatibility Notes, in the Release Guide*.

Introduction

Though the GCI interface is generic and may be used for executing tests in an almost arbitrary environment, there is often a significant effort required to get an adaptation up and running. This has prompted the development of a pre-built kernel which can be used to test executable test suites without requiring them to have complete adaptation layers for some platforms.

The TTCN Exerciser contains functionality for simulation of test suites without the requirement of having an SDL system or IUT to test. Furthermore it allows for one or several PCOs to communicate with a real IUT through a subset of the GCI interface. It is also prepared for adaptation with C++ and object oriented implementations under test by providing an “External C interface”.

Functionality Overview

The TTCN Exerciser provides several features that may come in handy for test development, execution and also adaptor production. The main features of the TTCN Exerciser are as follows:

- Simulated PCO communication

  Allows for a PCO not to be connected to an IUT. A message can be input in a PCO input queue at any time.

- Discrete time simulation

  Test the tester with user-defined timing. An explicit “timeout” command is available to allow for the next pending timer to expire at will. This mechanism has a timer resolution down to individual nanoseconds.

- Realtime simulation

  This mode uses real time as provided by the system clock of the host where the tester is executed. Practical limitations to the time resolution yields an accuracy of approximately 1 millisecond. The theoretical limit of the kernel is 1 nanosecond.

- TTCN test case validation

  Runs “random walks” of test cases in order to detect paths where no verdict is assigned, or where some input causes a deadlock, as well

as several other dynamic error conditions. This can be very useful for getting a degree confidence in the correctness and completeness of the test suites.

- Custom PCO definition

  Provides the ability to define custom PCOs for connection to real IUTs while still having access to all the TTCN Exerciser features.

- Source-level debugging of TTCN

  This includes setting breakpoints, highlighting lines in the TTCN Table Editor as it is running, and more.

- Concurrent TTCN implementation

  The kernel internally implements concurrent TTCN, meaning that there is no extra effort in running concurrent test cases.

- Dynamic MSC generation with the MSC generator

  For more information, see chapter 32, *The TTCN to C Compiler (in Windows)* and chapter 27, *The TTCN to C Compiler (on UNIX)*.

- Command scripting and logging

  Providing repeatability of simulated scenarios, as well as shortcuts for doing common operations.

- Dynamic error detection

  The TTCN Exerciser helps detect a number of dynamic error conditions that a static syntax and semantics analyzer will not detect.

Kernel Operation Modes

In order to appreciate the way the TTCN Exerciser operates, it is important to get an insight in the different global modes it may be in. Each of these modes are outlined in this section. The full global kernel state and behavior is a combination of these modes, and to get the desired behavior, it is good to know how to toggle between the modes at will.

Execution Modes

These are the three major modes of operation of the TTCN Exerciser:

- Simulation mode

  This mode has all PCOs simulated, and input is required from the test operator or user-defined scripts. In essence, there is no IUT and all messages that are sent are only indicated in the various test logs. All messages that are received must be specified by the test operator or a test execution script.

- Target execution mode

  This mode has all the PCOs defined by an adaptor writer (implemented in an adaptation effort). This mode provides source-level debugging of TTCN test suites and an option of discrete time simulation.

- Mixed mode

  This mode is available when some of the PCOs are connected to an actual implementation under test. It enables the features of both the simulation mode and the features of the target execution mode.

Timer Modes

These are the two timer modes:

- Realtime

  This mode uses the system clock to provide timing information. This is the normal mode to run in for target testers. In the case a tester is paused or a breakpoint is reached, the timer mode will be toggled to discrete time simulation. You can resume it by setting the timer mode to `realtime`. See "Realtime" on page 1365 for more information. The realtime mode is recognized by all log messages being prefixed with an asterisk (`*`).

- Discrete time simulation

  This mode only increments time when a `timeout` command is received. It is the default mode for simulations. In effect, the mode keeps track of running timers and their expiration time. This mode is automatically toggled to when the execution is stopped or paused, as well as for many of the kernel commands. See "Discrete" on page 1365 for more information. The discrete time simulation mode is recognized by all log messages being prefixed with a hyphen (`-`).

Control Modes

There are two control modes:

- Simulator UI control

  This mode is for using the SDL and TTCN Integrated Simulator user interface protocol in communications with the operator. The mode appends a set of tags to all messages that are emitted. It is assumed that the tester is started by the SDL and TTCN Integrated Simulator user interface, but the mode is also useful if logs are to be processed by other tools.

- Command line control

  This mode is for running in a command-line environment. An ETS built with the TTCN Exerciser can be run directly from a command-line, thereby providing means for automated, batch style testing.

PCOs

You can operate PCOs by either implicitly enqueueing messages in the PCO by using the receive command, or by customizing the PCO providing your own definition of the PCO behavior.

Customizing the Behavior of PCOs

The behavior of PCOs can be customized for achieving communications with “real” implementations under test. This is similar to the adaptation process. For more information, see chapter 36, *Adaptation of Generated Code*. The difference is that only the GciSend function and a polling/decoding function need to be implemented. Some familiarity with the GCI interface is required in order to successfully add a custom PCO implementation to the TTCN Exerciser – in particular, the use of the GciValue type should be known. Also, knowledge of the protocol and APIs for the implementation of the PCO is required.

Files

The TTCN Exerciser is interfacing to the code generator output in a manner that is conformant to the GCI interface recommendation. The `adaptor.c` file which is supplied with the TTCN Exerciser interface, is pretty much a plain adaptor that calls GCI equivalent functions in the

TTCN Exerciser. It is possible to edit the `adaptor.c` file while still retaining functionality of the TTCN Exerciser.

Custom PCO Registration

PCOs may be registered in the main function, before the IsmMain function is called. The PCO registration is done by calling the IsmRegisterPCOImplementation function. This functions arguments are described in detail in the `ism.h` file. Conceptually, each registered PCO is polled at a maximum interval supplied when the PCO is registered. The PCO has an associated polling function that should be used to check if a message is completely received on the PCO, and in that case decode it and use the GciReceive function to notify the TTCN runtime behavior of the newly arrived message. For sending, the PCO, registration function also requires a Send function argument. This function should send a message to the implementation under test.

It should be noted that these functions should all be possible to call from multiple threads, since the TTCN Exerciser may use multithreading to implement concurrent TTCN.

Timers

For both the realtime and the discrete time simulation, the implementation of the TTCN Exerciser has 64 bit timers. Time is kept as the difference from the timers expiration time and the last time the current time was updated. In realtime mode, the time is updated in the snapshot function. With discrete time simulation, the time is updated at the timeout command. With the 64 bit implementation and the constraint that the resolution is 1 ns, the consequences are that:

- No period of less than 1 ns can be measured.
- The maximum period that can be measured is more than 2 billion seconds, or some 65 years.

Hopefully, this resolution is sufficient for most applications. There are some aspects of discrete simulation that are well worth a special note:

- Discrete time simulation assumes that no processing time is required for executing TTCN and that there is no delay due to messaging and scheduling overhead in concurrent TTCN. This is not en-

tirely true. Apart from this, it is in many cases sufficient for testing a test script in a non-realtime environment.

- Discrete time simulation also expires timers at the exact right time (from the testers aspect, not in a realtime aspect). In many cases, a test process cannot be scheduled at exactly the time when a timer expires, and consequently there is normally a lag in a real-time tester. This lag should normally be accounted for by the test designer in determining timer durations.

Runtime Timer Errors and Warnings

With the timer implementation, there are also some conditions that may be reported in the conformance log:

- READTIMER of a timer that is not currently running. This condition will make the readtimer call return 0 and also produce a message in the conformance log.
- START of a timer with a ps resolution will result in the timer to get an approximate expiration time. This expiration time is the duration of the timer / 1000 ns. In effect this is too early. If the unit of a timer cannot be determined, it is assumed to be seconds. This condition should normally not appear, but will result in a message in the log and possibly also test case termination.

Test Suite Parameters

The test suite parameters are read at startup of the TTCN Exerciser. There are two locations where they are searched for:

- If the test suite parameters pics/pixit reference field contains the literal substring `.ttp`, the whole fields content will be treated as a file name, naming a file in which the parameter should be searched for. The file format is defined below.
- In a file named `pixit.ttp`.

In a pixit-file, the variables declared are to be named according to the following list of predefined type names. The names showed are typed with respect to case and spelling:

```
NULL     INTEGER BOOLEAN
BITSTRING       HEXSTRING       OCTETSTRING
NumericString   PrintableString TeletexString
```

```
T61String       VideotexString  VisibleString
ISO646String    IA5String       GraphicString
GeneralString   RTYPE           SEQUENCE
SEQUENCEOF      SET             SETOF
CHOICE          ENUMERATED      OBJECTIDENTIFIER
UTCTime         GeneralizedTime ObjectDescriptor
```

The file format is defined as:

```
File ::= {Line};
Line ::= Comment | ParameterNameAndValue;
Comment ::= <Non alpha Character> <Any text> | <empty line>;
ParameterNameAndValue ::= ParameterName ISMValueEncoding;
```

The ISMValueEncoding is defined in a separate section. An example would be:

```
#
# This is my parameter file, named pixit.ttp, located in the
# working directory of the ETS with the TTCN Exerciser.
#

TspFoo          3
TspUsePDUPDUA ( INTEGER 1, BOOLEAN TRUE )

# End of this example file.
```

Test Case Validation

The TTCN Exerciser has a rudimentary automatic test validation feature in that it supports running tests with random inputs and timeouts. This feature may be used to attempt to detect missing alternatives and some runtime error conditions without having to manually step through all these alternatives.

Detected conditions include:

- Test incompleteness
- Tests failing to terminate
- Concurrent TTCN dynamic errors:
  - Failure to terminate a PTC
  - Creation of already started PTC
- Tests with paths missing a verdict
- A number of other conditions such as timer errors etc.

The random walk does not provide a guaranteed detection of these conditions, but it will run thousands of test tests in the time it normally takes to run one test.

The random walk has no feature for message generation, and hence requires messages to be imported or defined in the tool. All the messages are stored in a list of eligible messages. This list is referred to as the message list.

The random walk selects random messages from this list, and also timeouts if in a given state, any timers are running. This is repeated until a verdict is reached, an deadlock is detected, or until a defined maximum depth is reached.

Reports are stored in a report list and can be saved or reviewed.

Also, test statistics are stored while running the random walk. These test statistics can be used to determine how many tests were run in total, and which verdicts were reached.

TTCN Exerciser Commands

The TTCN Exerciser operates through a command line interface (which can be encapsulated by for instance the SDL and TTCN Integrated Simulator user interface).

Example 223: General syntax of the command input

```
CommandInput::= {CommandLine} EOF
CommandLine::= Comment|Command
Comment::= <Non Alpha Character> <Free Text>
Command::= <Recognized command or '+'> <Parameters> <NewLine>
```

It is permitted to have any amount of spaces or tabs before the command start. In general, the assembler code modes of some editors works quite well for composing command files.

Example 224: A command file

```
;;;
;;; File Name: example
;;; This is an example command file, uses ; for command delim.
;;;
```

```
;; Initialisation (see command descriptions below)
cl                      ; Set command line log mode
loglevel 2      ; Set log level to 2
nomsc                   ; Disable MSC generation
nopoll                  ; Disable command polling
discrete        ; Use discrete time sim

;; Now run a test case
step TestCase1  ; Will create context
receive Lower ConnectInd { Address 3 }
run                     ; Will actually run till idle
timeout                 ; Do a timeout when idle

;;; End of this file...
```

These are the available commands when a tester built with the TTCN Exerciser is run from the command line or with a batch script. The commands may be abbreviated, as long as the abbreviation yields one unique command name. For instance, the command `cancel` may be abbreviated with `ca` but not with `c`, since that would also match the `cmdlog` command.

The command listing below has the following information:

- Command name

  The formal name of the command, as defined in the ETS/SIMUI protocol document, available on request from IBM Rational.

- Aliases

  Alternative command names that may be handy if running from a command line interface. These are often common names for similar operations in debuggers. A hyphen indicates that the command has no aliases apart from any possible abbreviations. Aliases may also be abbreviated, for instance, the alias `rts` for the `realtime` command can be abbreviated `rt`.

- Synopsis

  The command with “symbolic” parameters. If a parameter is surrounded by square brackets “`[ ]`”, it is optional. If a parameter is surrounded by curly brackets “`{ }`”, it indicates that a list may be accepted. If the parameter is surrounded by angle brackets “`< >`”, it means that the name should be substituted by an identifier of an object of an appropriate type.

- Description

  An informal textual description of the command and some of its possible side-effects.

- Example

  An example of the command used in some context. The examples frequently use other commands than the described one in order to explain the use of the command.

General Commands

Help

- Command name: `help`
- Aliases: `-`
- Synopsis: `help`
- Description:

  List the available commands with a brief note on each commands purpose. The general format is:

  `help [SR *]` Display a brief help on commands

  The printed records are:

  `<command-name> [<states> <debug>] <description>`

`<command-name>`	The unique name of the command.
`<states>`	The set of states where the command is applicable: • `S` – Stopped, no test case is running. • `R` – Running, as test case is in progress.

<debug>	Denotes if the tester needs to be generated with the line information for the command to be running correctly: • D – Requires line-number information. • * – Available with or without line-number information.
<description>	A brief summary of the command functionality.

The help command will automatically transition the tester to a Paused state if it is in running state.

Example 225 ----------

```
-       Command?
help
```

Quit

- Command name: quit
- Aliases: -
- Synopsis: quit
- Description:

Terminate the current test case and also terminate the tester program.

Example 226 ----------

```
-       Command?
quit
```

Include

- Command name: include
- Aliases: -

- Synopsis: `include <filepath>`
  `+filepath`
- Description:

  Enqueue all the commands from the file named by file path. The file path can be relative to the current working directory of the tester, or it may be absolute. Also, the shortcut `+` may be used for simpler access to common scripts. It is recommended but not required that scripts have a file extension of `.ics` (ISM Command Script).

  It is sometimes desirable to use the command `nopoll` in conjunction with script reading, in particular when running test cases from scripts. The `nopoll` command gives a more synchronous behavior.

Example 227 ----------

```
-       Command?
include setup.ics
-       Command?
+setup.ics
```

Test Management Commands

List

- Command name: `list`
- Aliases: `-`
- Synopsis: `list`
- Description:

  List all available test cases. Test cases whose selection expression evaluates to false, get the tag `[N/S]` appended to their name to show that the case is not selected.

Example 228 ----------

```
        Command?
list
MyTest1
MyTest2
```

```
MyTest3[N/S]
```

Glist

- Command name: `glist`
- Aliases: `-`
- Synopsis: `glist`
- Description:

  List all available test case groups.

Example 229 ----------

```
-       Command?
glist
MyGroup1
MyGroup2
```

Run

- Command name: `run`
- Aliases: `go`, `start`
- Synopsis: `run {<TestCaseOrGroupNames>}`
- Description:

  Start the execution of the named test cases or groups. The test cases will run until a breakpoint is reached, the test case terminates, or until another command is entered.

  If a test is already in progress, this command will continue the test execution until one of the above conditions are met.

Example 230 ----------

```
-       Command?
run MyTest1
-
```

```
-       Line: MyTest1 1
-       Line: MyTest1 2
-       Line: MyTest1 3
-       Line: MyTest1 4
-       At breakpoint MyTest1 5
-       Line: MyTest1 5
-       Command?
run
```

Step

- Command name: `step`
- Aliases: `line`
- Synopsis: `step {<TestCaseOrGroupNames>}`
- Description:

  Start the execution of the named test cases or groups. The test cases will run until a breakpoint is reached, the test case terminates, or until another command is entered.

  As opposed to the run command, the test execution will also stop if a line is matched, or if a snapshot (idle) state is reached. This command requires that the tester has been built with line debug information for correct operation.

  If a test is already in progress, this command will continue the test execution until one of the above conditions are met.

  This command will toggle the timer mode to Discrete, since it is not possible to use real-time and line stepping in conjunction.

  Example 231 ––

```
-       Command?
step MyTest1
-
-       Line: MyTest1 1
-       Command?
step
-       Line: MyTest1 2
-       Command?
step
-       Line: MyTest1 3
-       Command?
```

Stop

- Command name: `stop`
- Aliases: `pause`, `break`
- Synopsis: `stop`
- Description:

  Pause the execution. This will toggle the time mode to discrete and halt the test case without discarding any context information. This command requires line debug information to be included when the ETS is built to function correctly.

Example 232

```
-       Command?
run MyTest1
-       Line:   MyTest1 1
-       Line:   MyTest1 2
stop
-       Command?
step
-       Line:   MyTest1 3
-       Command?
```

Cancel

- Command name: `cancel`
- Aliases: `abort`
- Synopsis: `cancel`
- Description:

  Cancel the current test run and reset the tester. This leaves the IUT in an undefined state. It is not recommended to use this command unless a test has lost contact with the IUT or is incomplete.

Example 233 ----------

```
- run test1
```

Loglevel

- Command name: `loglevel`
- Aliases: `ll`
- Synopsis: `loglevel [0-3]`
- Description:

  This command defines how much conformance logging should be performed. The levels are:

`0`	Only verdicts
`1`	Level 0 + PTCs, PCOs and TIMERS (default level)
`2`	Level 1 + matched lines and test trees
`3`	Level 2 + non-matched lines and defaults

If no level is supplied, the default level will be used.

Example 234 ----------

```
-       Command?
loglevel 3
-       Set conformance log level = 3
-       Command?
```

Savestats

- Command name: `savestats`
- Aliases: `ss`
- Synopsis: `savestats [<file path>]`

- Description:

  Save a simple tabulated file with a list of what tests have been executed and what their verdicts were. This is useful for tracking test results over time, typically by loading the result file in some type of information management program. If the file path is omitted, the default file path `results.txt` will be used.

Example 235 ----------

```
-       Command?
savestats testresults.txt
-       Command?
# Just list the contents of the file
system cat testresults.txt
Test                            Verdict
MyTest1                         PASS
MyTest2                         FAIL
MyTest1                         INCONCLUSIVE
```

Liststats

- Command name: `liststats`
- Aliases: `ls`
- Synopsis: `liststats`
- Description:

  List the test results of all test cases that have been run since the TTCN Exerciser was started, or since the last clearstats command.

Example 236 ----------

```
-       Command?
liststats
- Listing test statistics:
- MyTest1                       PASS
- MyTest2                       FAIL
- MyTest1                       INCONCLUSIVE
-       Command?
```

Clearstats

- Command name: `clearstats`
- Aliases: `cs`
- Synopsis: `clearstats`
- Description:

  Clear the list of test results

Example 237 ----------

```
-       Command?
clearstats
-       Clearing test statistics.
-       Command?
```

Test Debugging Commands

Most of these test debugging commands require the tester to be built with line debugging information for correct operation.

Breakpoints

- Command Name: `breakpoints`
- Aliases: `bps`
- Synopsis: `breakpoints`
- Description:

  List all currently set breakpoints.

Example 238 ----------

```
-       Command?
breakpoints
MyTest1                 5
-       Command?
```

Breakpoint

- Command name: `breakpoint`
- Aliases: `bp`
- Synopsis: `breakpoint [table] [line]`
- Description:

  Set a breakpoint at line in table. If either is omitted, the last matched line or table will be used respectively.

Example 239 ----------

```
-       Command?
breakpoints
-       Command?
step MyTest1
-       Line: MyTest1 1
-       Command?
breakpoint
-       Breakpoint set at MyTest1 1
-       Command?
breakpoint 4
-       Breakpoint set at MyTest1 4
-       Command?
breakpoint MyTest2 2
-       Breakpoint set at MyTest2 2
breakpoints
-       MyTest1 1
-       MyTest1 4
-       MyTest2 2
-       Command?
```

Delete

- Command name: `delete`
- Aliases: `-`
- Synopsis: `delete [table] [line]`
- Description:

  Delete the breakpoint at table and line. If either or both is omitted, the last matched line will be used.

Example 240 ----------

```
-       Command?
breakpoint MyTest1 3
-       Breakpoint set at MyTest1 3
-       Command?
breakpoint MyTest1 5
-       Breakpoint set at MyTest1 5
run
-       Line:   MyTest1 1
-       Line:   MyTest1 2
-       Breakpoint reached: MyTest1 3
-       Line:   MyTest1 3
-       Command?
delete
-       Breakpoint deleted: MyTest1 3
-       Command?
delete MyTest1 5
-       Breakpoint deleted: MyTest1 5
-       Command?
breakpoints
-       Command?
```

Disable

- Command name: `disable`
- Aliases: -
- Synopsis: `disable`
- Description:

  Temporarily disable all breakpoints.

Example 241 ----------

```
-       Command?
disable
-       Breakpoints disabled
-       Command?
```

Enable

- Command name: `enable`

- Aliases: -
- Synopsis: `enable`
- Description:

  Re-enable breakpoints after a disable command.

Example 242 ----------

```
-       Command?
disable
-       Breakpoints disabled
-       Command?
enable
-       Breakpoints enabled
-       Command?
```

Timers

- Command name: `timers`
- Aliases: -
- Synopsis: `timers`
- Description:

  List all currently running timers. The output format contains the following information columns:

`PTC`	The PTC to which the timer belongs
`Timer Name`	The TTCN name of the timer
`Id`	The GCI timer identifier
`Remaining (s)`	The remaining time to timeout in seconds

Example 243 ----------

```
-       Command?
timers
-       PTC     Timer Name      Id      Remaining (s)
-       MTC     Tms                     210         0.400000000
-       MTC     TWatchDog       211       59.600000000
```

```
-       PTC1    Tms                         214         0.400000000
-
-       Command?
```

Pcos

- Command name: `pcos`
- Aliases: -
- Synopsis: `pcos`
- Description:

  List all PCOs and the contents of their associated queues.

Example 244

```
-       Command?
step MyTest1
-       Line: MyTest1 1
-       Command?
receive LowerPCO ConnectReq { 6 "Peer1" }
-       Command?
receive LowerPCO ConnectReq { 7 "Peer2" }
-       Command?
pcos
-       PCO UpperPCO
-            <empty input queue>
-       PCO LowerPCO
-            1:  ConnectReq { 6 "Peer1" }
-            2:  ConnectReq { 7 "Peer2" }
-
-       Command?
```

Ptcs

- Command name: `ptcs`
- Aliases: -
- Synopsis: `ptcs`
- Description:

List all the currently active test components. The output columns include the following information:

`Name`	The name of the test component
`Id`	The GCI id of the test component
`Thr`	The host operating system thread id of the component
`Table`	The table name of the last matched TTCN line
`Line`	The last matched line number

Also, the currently active test component will be indicated with an arrow (`->`) in the output of this command.

Example 245 ----------

```
-       Command?
ptcs
-       Name            Id      Thr             Table Line
-       MTC             124     0             MyTest1 5
-    -> PTC1            125     13            MyStep1 3
-
-       Command?
```

Gett

- Command name: `gett`
- Aliases: `print`
- Synopsis: `gett [<ptc>] <variablename>`
- Description:

  Print the value of the named variable in the named test components-context. If the component name is omitted, the current component will be used.

  Example 246 ----------

```
-       Command?
gett MTC TsvFoo
```

```
-       TsvFoo: 4711
-       Command?
gett PTC1 TsvFoo
-       TsvFoo: 1234
-       Command?
```

Test Simulation Commands

Discrete

- Command name: `discrete`
- Aliases: `dts`
- Synopsis: `discrete`
- Description:

  Toggle to discrete time simulation mode. The command has no effect if already in discrete time simulation mode.

  The log prefix will be switched to `'-'`.

Example 247

```
*       Line:   Table1 3
*       Line:   Table1 4
*       Line:   Table1 5
discrete
*       Discrete time simulation is now used.
-       Command?
```

Realtime

- Command name: `realtime`
- Aliases: `rts`
- Synopsis: `realtime`
- Description:

  Toggle to realtime simulation mode. The command has no effect if already in realtime simulation mode.

The log prefix will be switched to `'*'`. A running test will immediately resume execution.

Example 248 ----------

```
-       Line:   Table1 1
-       Command?
step
-       Line:   Table1 2
realtime
-       Realtime simulation is now used
*       Line:   Table1 3
*       MTC             SEND    ASP1
*       MTC             START   Tms(200)
*       Line:   Table1 4
*       MTC             TIMEOUT Tms
*       Line:   Table1 5
*       FINAL VERDICT: PASS
*       Command?
```

Timeout

- Command name: `timeout`
- Aliases: `to`
- Synopsis: `timeout`
- Description:

  Switch to discrete time simulation. Force the shortest pending timer to expire.

Example 249 ----------

```
-       Command?
realtime
*       Command?
run Test1
*       Line:   Test1 1
*       MTC             START TWatchDog(60)
timeout
-       Discrete time simulation is now used
-       Command?
run
-       Line:   Default1 1
-       MTC             TIMEOUT TWatchDog
```

```
-       FINAL VERDICT:  FAIL
-       Command?
```

Receive

- Command name: `receive`
- Aliases: `input`
- Synopsis: `receive <PCOId> <ISMValueEncoding>`
- Description:

  Enqueue the message defined by ISMValueEncoding in the PCO input queue connected to the named PCO.

  It is possible to use input values of types SEQUENCE OF, SET and SET OF. The left/right braces syntax should be:

  – SEQUENCE - '{' and '}'

  – SEQUENCE OF - '[' and ']',

  – SET - '{.' and '.}'

  – SET OF - '[.' and '.]'

  The field's names are mandatory for SET values and optional for SEQUENCE values.

  For an alternative form, see “Messageinput” on page 1376.

Example 250 ----------

```
-       Command?
pcos
-       PCO UpperPCO
-           <empty input queue>
-       PCO LowerPCO
-           <empty input queue>
-
-       Command?
receive UpperPCO AcceptCall { 1, TRUE, { 34, 13, 99, '5'B } }
-       Command?
pcos
-       PCO UpperPCO
-           1:  AcceptCall { 1, TRUE, { 34, 13, 99, '5'B } }
```

```
-       PCO LowerPCO
-           <empty input queue>
-       Command?
```

MSC Generation Commands

Nomsc

- Command name: `nomsc`
- Aliases: -
- Synopsis: `nomsc`
- Description:

  Disable MSC generation.

  The command is only available when the tester is stopped.

Example 251

```
-       Command?
nomsc
-       MSC generation disabled
-       Command?
```

Decomposed

- Command name: `decomposed`
- Aliases: -
- Synopsis: `decomposed`
- Description:

  Enable MSC generation with decomposed MSCs. For more information, see chapter 32, *The TTCN to C Compiler (in Windows)* and chapter 27, *The TTCN to C Compiler (on UNIX)*.

  The command is only available when the tester is stopped.

Example 252 ----------

```
-       Command?
decomposed
-       Decomposed MSC generation enabled
-       Command?
```

Composed

- Command name: `composed`
- Aliases: -
- Synopsis: `composed`
- Description:

  Enable MSC generation with composed MSCs. For more information, see chapter 32, *The TTCN to C Compiler (in Windows)* and chapter 27, *The TTCN to C Compiler (on UNIX)*.

  The command is only available when the tester is stopped.

Example 253 ----------

```
-       Command?
composed
-       Composed MSC generation enabled
-       Command?
```

Mscsystem

- Command name: `mscsystem`
- Aliases: -
- Synopsis: `mscsystem [<system-name>]`
- Description:

  Define the name of the IUT instance in the composed MSC generation. This is for making it easier to generate MSCs which should later be reused in a system verification with an MSC verification tool

such as the SDL Explorer, as well as for test re-generation with a tool such as Autolink.

If the system-name parameter is omitted, a default system name will be used (by default `IUT`).

Example 254 ----------

```
-       Command?
mscsystem env_0
-       IUT Name for composed MSC generation defined.
-       Command?
```

Mscprefix

- Command name: `mscprefix`
- Aliases: `-`
- Synopsis: `mscprefix <path/prefix>`
- Description:

  Set a prefix for file names used when generating MSCs. This prefix is prepended to the file name and can be both a path or a file prefix.

  The default is `log_`.

Example 255 ----------

```
-       Command?
mscprefix /tmp/traces/trace_
-       Path/file prefix defined for MSC generation.
-       Command?
#
#       All test traces will be saved in the named directory
#
```

Test Validation Commands

Clearreports

- Command name: `clearreports`
- Aliases: `cr`
- Synopsis: `clearreports`
- Description:

  Clear the list of reports of dynamic errors encountered so far in test runs.

Example 256 ----------

```
-       Command?
clearreports
-       Reports cleared.
-       Command?
```

Listreports

- Command name: `listreports`
- Aliases: `lr`
- Synopsis: `listreports`
- Description:

  List all the reports of dynamic errors encountered so far in test runs. The reports are in the following general format:

  `Report <index>: [<table>] <line>: <description>`

  The `<table>` field is optional. If it is not present, it implies that the report was not detected in the context of a running test component. If the `<line>` field is 0, there is a context available, but the context is not executing.

  Also, if no line debug information is available, the table and line will be omitted and 0 respectively.

Example 257 ----------

```
-       Command?
listreports
-       Listing reports:
-       Report  1:      TestCase1 7: Missing ?DONE in MTC
-       Report  2:      TestCase2 3: Incomplete test case
-       Command?
```

Savereports

- Command name: `savereports`
- Aliases: `sr`
- Synopsis: `savereports [<filename>]`
- Description:

  Save the reports to named file, or to the file `reports.txt` if no file name is supplied. The file format is the same as for the `listreports` command output.

Example 258 ----------

```
-       Command?
listreports
-       Listing reports:
-       report  1:      TestCase1 7: Missing ?DONE in MTC
-       report  2:      TestCase2 3: Incomplete test case
-       Command?
```

Messagefile

- Command name: `messagefile`
- Aliases: `mf`
- Synopsis: `messagefile <filename>`
- Description:

Append the message definitions in the messagefile to the list of input messages that can be used with the `messageinput` command, and also for selection of messages with the `randomwalk` command.

The syntax of the file named by `filename` is:

```
MessageFile ::= MessageLine | CommentLine;
MessageLine ::= PcoId ISMValueEncoding NL;
CommentLine ::= NonAlphaChar Any NL;
```

Example message file:

```
# This is an example message file named file1.msg

PCO1 ASP1 { 1 2 FALSE '1234'O }
PCO2 ASP3 { 4 2 TRUE '3123'H }

# end of example file
```

Also note that messages can be added with the command `messageadd`. The messagefile is mainly intended for message lists generated by other means than with this tool.

Example 259 ––

```
-       Command?
messagefile file1.msg
-       Adding message definitions:
-       PCO1 ASP1 { 1 2 FALSE '1234'O }
-       PCO2 ASP3 { 4 2 TRUE '3123'H }
-       Command?
```

––

Messageadd

- Command name: `messageadd`
- Aliases: `ma`
- Synopsis: `messageadd <pconame> ISMValueEncoding`
- Description:

  Add the named message on the named PCO to the inputs that can be sent with the `messageinput` command and from which inputs are selected with the random walk.

It is possible to use input values of types SEQUENCE OF, SET and SET OF. The left/right braces syntax should be:

– SEQUENCE - '{' and '}'

– SEQUENCE OF - '[' and ']',

– SET - '{.' and '.}'

– SET OF - '[.' and '.]'

The field's names are mandatory for SET values and optional for SEQUENCE values.

Example 260 ––

```
-       Command?
messagelist
-       Listing message definitions
-       1   : PCO1 ASP1 { 1 2 FALSE '1234'O }
-       2   : PCO2 ASP3 { 4 2 TRUE '3123'H }
-       Command?
messageadd PCO1 ASP2 ( FALSE TRUE )
-       Command?
messagelist
-       Listing message definitions
-       1   : PCO1 ASP1 { 1 2 FALSE '1234'O }
-       2   : PCO2 ASP3 { 4 2 TRUE '3123'H }
-       3   : PCO1 ASP2 ( FALSE TRUE )
-       Command?
```

––

Messagelist

- Command name: `messagelist`
- Aliases: `ml`
- Synopsis: `messagelist`
- Description:

  List all messages defined in the message list.

  Example 261 ––

  ```
  messagelist
  ```

```
-       Listing message definitions
-       1   : PCO1 ASP1 { 1 2 FALSE '1234'O }
-       2   : PCO2 ASP3 { 4 2 TRUE '3123'H }
-       3   : PCO1 ASP2 ( FALSE TRUE )
-       Command?
```

Messageclear

- Command name: `messageclear`
- Aliases: `mc`
- Synopsis: `messageclear [<index>]`
- Description:

  Clear the message at index `<index>` in the message list. If no index is provided, all messages are cleared. The `messagelist` command prints indices for each message definition.

  Also note that the `messageclear` command may change the ordering and indices of messages at higher indices.

Example 262 ----------

```
-       Command?
messagelist
-       Listing message definitions
-       1   : PCO1 ASP1 { 1 2 FALSE '1234'O }
-       2   : PCO2 ASP3 { 4 2 TRUE '3123'H }
-       3   : PCO1 ASP2 ( FALSE TRUE )
-       Command?
messageclear 1
-       Cleared message definition.
-       Command?
messagelist
-       Listing message definitions
-       1   : PCO2 ASP3 { 4 2 TRUE '3123'H }
-       2   : PCO1 ASP2 ( FALSE TRUE )
-       Command?
messageclear
-       Clearing all message definitions
-       Command?
messagelist
messagelist
-       Listing message definitions
-       Command?
```

Messageinput

- Command name: `messageinput`
- Aliases: `mi`
- Synopsis: `messageinput [<index>]`
- Description:

  Provide the message at index `<index>` in the message list as input to the Tester. This is a short form of the `receive` command, appropriate for frequently used messages, and it may also be used to test contents of message files.

  The message is decoded and added at the end of the named PCO queue.

Example 263 ----------

```
-       Command?
messagelist
-       Listing message definitions
-       1  : PCO2 ASP3 { 4 2 TRUE '3123'H }
-       2  : PCO1 ASP2 ( FALSE TRUE )
-       Command?
messageinput 2
-       Command?
step
-       MTC     RECEIVE         PCO1 ? ASP2
-       MTC     STARTTIMERTimer1
```

Randomwalk

- Command name: `randomwalk`
- Aliases: `rw`
- Synopsis: `randomwalk [<repetitions>] {<testcaselist>}`
- Description:

  Will run the test cases of `<testcaselist>` `<repetitions>` times, providing random inputs and timeouts as they go. Reports are collected while running the test cases. Test logging is disabled, but

can be re-enabled if a breakpoint is reached while doing the random walk.

Example 264 ----------

```
-       Command?

# This is a command for running the test case MyTestCase 100
# times while providing random inputs from the message list
# and also timeouts.

randomwalk 100 MyTestCase

-       Starting random walk...

[some logging of verdicts and selected path]

-       Random walk completed
-       Command?
listreports
-       Listing reports:
-       report  1: MyTestCase 4: READTIMER: Timer not running
-       Command?
```

Maxdepth

- Command name: `maxdepth`
- Aliases: -
- Synopsis: `maxdepth <depth>`
- Description:

  Define how many messages and timeouts a test case may receive, until a test case run not producing a verdict should be abandoned. If the maxdepth is exceeded, a report will be produced.

  Example 265 ----------

  ```
  -       Command?
  maxdepth 10
  -       Defined max depth for random walk
  -       Command?
  ```

Kernel Management Commands

Cl

- Command name: `cl`
- Aliases: -
- Synopsis: `cl`
- Description:

  Turn off the SDL and TTCN Integrated Simulator user interface tagging. This makes the outputs of the tester more human-readable. By default, this tagging is enabled, so the first command you enter when running from the command line is typically a `cl` command.

Example 266 ----------

```
<VERDICT>
-       FINAL VERDICT:PASS
</VERDICT>
<MESSAGE UI::READY>
-       Command?
</MESSAGE UI::READY>
cl
-       Output tagging disabled
-       Command?
list
-       TestCase1
-       TestCase2
-       Command?
```

Ui

- Command name: `ui`
- Aliases: -
- Synopsis: `ui`
- Description:

  Enable the SDL and TTCN Integrated Simulator user interface tagging. This makes the output less readable for a human, but easier to

parse by computer programs (such as the SDL and TTCN Integrated Simulator user interface).

Example 267 ----------

```
cl
-       Output tagging disabled
-       Command?
list
-       TestCase1
-       TestCase2
-       Command?
ui
<LOG>
-       Using simulator ui output tagging
</LOG>
<MESSAGE UI::READY>
-       Command?
</MESSAGE UI::READY>
```

Poll

- Command name: `poll`
- Aliases: `-`
- Synopsis: `poll`
- Description:

  Enable command polling and dispatch on each matched TTCN line. This slows down the execution somewhat, but makes it possible to interrupt a tester in other states than an idle state. The command requires that the tester was built with line debugging enabled.

Example 268 ----------

```
-       Command?
poll
-       Command?
```

Nopoll

- Command name: `nopoll`
- Aliases: -
- Synopsis: `nopoll`
- Description:

  Disable command polling in other states than stopped or snapshot (idle). This increases the execution speed, and also facilitates script writing by ensuring that a tester is in one of these two known states before it processes its next command.

Example 269 ----------

```
-       Command?
nopoll
-       Command?
```

Cmdlog

- Command name: `cmdlog`
- Aliases: -
- Synopsis: `cmdlog [<filepath>]`
- Description:

  Start or stop a command log. If the `<filepath>` argument is supplied, the named file will be opened for writing and all commands from the command line or included scripts will be written to the file. If no file name is supplied, the file will be closed and command logging disabled.

Example 270 ----------

```
-       Command?
cmdlog  asp1
-       Command?
receive pco1 asp1 { '10'B FALSE '11'B }
-       Command?
run
```

```
-       Command?
cmdlog
-       Command?
+asp1
-       Command?
+asp1
-       Command?
pcos
-       PCO pco1
-           1: asp1 { '10'B FALSE '11'B }
-           2: asp1 { '10'B FALSE '11'B }
-           3: asp1 { '10'B FALSE '11'B }
```

Status

- Command name: `status`
- Aliases: -
- Synopsis: status
- Description:

  Display current TTCN Exerciser status and mode information overview.

Example 271 ----------

```
-       Command?
status
-       Tester status: STOPPED
-       Time Mode:     Discrete Time Simulation
-       Logging level: 2
-       Breakpoints:   Disabled
-       Command Log:   Disabled
-       Logging Mode:  Command Line
-
-       Command?
```

System

- Command name: `system`
- Aliases: -
- Synopsis: `system <commandline>`

- Description:

  Execute the <commandline> using the command line interpreter of the platform. Uses the C stdlib function system(<commandline>). Use this with care for reviewing, deleting and creating files.

Example 272 ----------

```
-       Command?
system mkdir ./msctraces
-       Command?
mscprefix ./msctraces/trace_
-       Command?
composed
-       Command?
run tc1 tc2 tc3 tc4

        [snip]

-       Command?
system ls ./msctraces
trace_tc1.mprtrace_tc2.mprtrace_tc3.mprtrace_tc4.mpr
-       Command?
```

Log

- Command name: log
- Aliases: -
- Synopsis: log [<filepath>]
- Description:

  Start or stop a total log. If the <filepath> argument is supplied, the named file will be opened for writing. All commands from the command line or included scripts, conformance log and messages from ETS will be written to the file. If no file name is supplied, the file will be closed and logging disabled.

Example 273 ----------

```
-       Command?
log mylog.txt
```

```
-       Command?
receive pco1 asp1 { '10'B FALSE '11'B }
-       Command?
run
-       Command?
log
```

ISM Value Encoding

The ISM Value Encoding defines how values are encoded when sent, or decoded when received. It also defines the format for values when printed or read as test suite parameters. The format is not 100% compatible with the SDL Simulator format, or the MSC generation format, though in most cases it will be able to read those values as well.

The general syntax for the value notation is as follows. Rules named with all capital letters are tokens.

```
ISMValueEncoding ::= OptionalTypeAndValue ;
OptionalTypeAndValue ::= [TYPE] Value ;
TYPE ::= <Name of type defined in TTCN test suite> ;
Value ::= Composite | Atomic | OMIT;
Atomic ::= CSTRING | OSTRING | HSTRING | BSTRING | INTEGER | BOOLEAN ;
Composite ::= LPAR {OptionalTypeAndValue OPTCOMMA } RPAR;
CSTRING  ::= '"' .* '"' ;
OSTRING  ::= ''' ([09afAF][09afAF])* ''O' ;
HSTRING  ::= ''' [09afAF]* ''H' ;
BSTRING  ::= ''' [01]* ''B' ;
INTEGER  ::= '-' [09]* | [09]* ;
BOOLEAN  ::= 'TRUE' | 'FALSE' | 'true' | 'false' ;
LPAR     ::= '{' | '(' ;
RPAR     ::= '}' | ')' ;
OPTCOMMA ::= ',' | ;
OMIT     ::= '-';
```

The CSTRING type is by default IA5String, use TYPE to override with a different character string type. Note also that in TTCN, the character string types are compatible and it may not be necessary to make the distinction.

Examples of this encoding include:

Encoding	Description
`1`	The INTEGER value 1
`FALSE`	The BOOLEAN value FALSE

Encoding	Description
`MyINTEGER 23`	The INTEGER derivate MyINTEGER value 23
`BIT4 '1001'B`	The BITSTRING derivate BIT4 value ‘1001’B
`{ 1 , 2 , 3 }`	A composite type with field values 1, 2 and 3
`ASP1 { 711, TRUE }`	An ASP1 with field 1 = 711 and field 2 = TRUE
`ASP2(BIT4 '1001'B PDU1(12))`	An ASP2 with second field being a PDU1

Chapter

35

TTCN Test Suite Generation

This chapter describes two ways for generating TTCN test suites based on SDL specifications. The first one is to use TTCN Link, which assists you in manual specification of test suites. The other one is to use the Autolink feature of the SDL Explorer, which allows automatic generation of test suites.

For more information about the SDL Explorer, see *chapter 52, The SDL Explorer*.

Tutorials for TTCN Link and Autolink can be found in chapter 8, *Tutorial: The TTCN Link* and chapter 9, *Tutorial: The Autolink Tool, in the TTCN Suite 6.2 Getting Started*.

Introduction

Testing is one of the most important steps in the development of a new product. Often, it is also very time consuming and costly. As part of the Conformance Testing Methodology and Framework, the *Tree and Tabular Combined Notation* (TTCN) has been defined as a formal language for test suite specification. A test suite consists of four basic parts: The test suite overview, the declarations, the constraints and the dynamic behavior description.

In the TTCN Suite, TTCN test suite generation is supported by TTCN Link and Autolink. They both use an SDL specification as the basis for test generation, but they differ in their functionality.

TTCN Link generates the TTCN declarations part automatically and you use it for interactive building of test cases in the dynamic part.

Autolink is embedded in the SDL Explorer. In addition to the SDL specification, it uses MSCs for test purpose descriptions. With this input, Autolink generates the declarations, constraints and dynamic behavior description parts of a TTCN test suite automatically.

In comparison, the test generation features of Autolink are superior to the ones of TTCN Link. If for some reason the test purpose description with MSCs is not applicable, then you should use TTCN Link. In any case, test cases built with TTCN Link can be merged with test cases generated by Autolink.

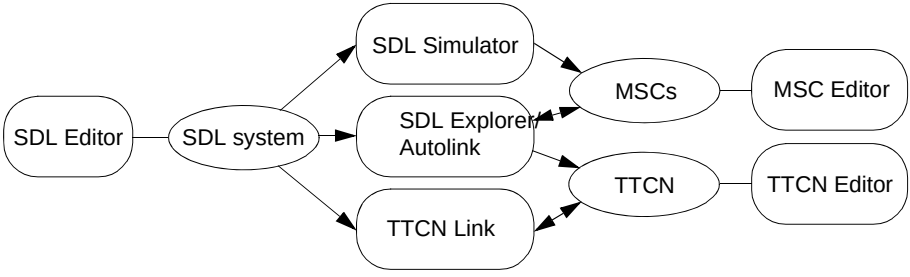


Figure 235: Overview of TTCN Link and Autolink

TTCN Link – Generation of Declarations

TTCN Link automatically generates the TTCN declarations part based on an SDL specification. The default dynamic behavior table is also generated. It contains timeout and otherwise statements for each PCO, which will ensure that any incorrect response from the implementation under test always will give a FAIL verdict as a result from the test case. After the default and constraints tables have been generated, you can interactively build test cases.

When you use TTCN Link, there are four (five) phases involved:

1. In the SDL Editor, you prepare an SDL specification.
2. In the Organizer, you generate a TTCN Link application.
3. In the TTCN Suite, you use TTCN Link for generating the declarations part.
4. In the TTCN Suite, you interactively build test cases.
5. Optionally, you may also merge the test suite with a TTCN-MP file, possibly generated by Autolink.

For more information about TTCN Link, see "Using TTCN Link" on page 1389.

Autolink – Generation of a Test Suite

Autolink can be used for automatic generation of TTCN test suites based on an SDL specification and a number of MSCs. The steps involved when you use Autolink are:

1. In the SDL Editor, you specify the SDL system to be used.
2. In the Organizer, you generate an SDL Explorer.
3. In the SDL Explorer, you define a number of traces through the SDL system for which you want to derive test cases. Each trace is stored as an MSC. Alternatively, you may create the MSCs manually in the MSC Editor or generate them with the help of the SDL Simulator.
4. In a text editor, you define an Autolink configuration. The configuration tells Autolink how to map SDL signals and signal parameters onto TTCN constraint names, and how to group test cases and test steps.

5. In the Explorer, you generate intermediate representations of the test cases and constraints from the MSCs. This can either be done by state space exploration or by direct translation to TTCN.
6. In the Explorer, you may modify the generated constraints. Afterwards, the result is saved in a TTCN-MP file.
7. The TTCN-MP file can be opened in the TTCN Suite, and to complete the test suite, the test suite overview has to be generated. **On UNIX**, you have to generate it explicitly. **In Windows**, the overview is generated automatically, for example before you print.

For more information about Autolink, see "Using Autolink" on page 1431.

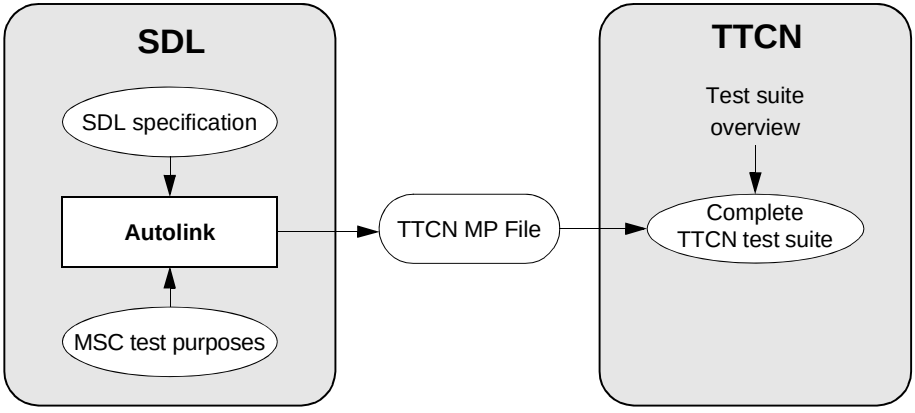


Figure 236: Generation of a TTCN test suite with the TTCN Suite and Autolink

Using TTCN Link

The phases involved when you are using TTCN Link – "Preparing for the Generation of Declarations" on page 1389, "Generating the Declarations" on page 1390 and "Creating Test Cases" on page 1396 – will be described below. You can also read about "Showing SDL System Information" on page 1398 and "Merging TTCN Test Suites in the TTCN Suite" on page 1398.

Preparing for the Generation of Declarations

Before it is possible to generate the declarations, you have to do two things:

1. Adapt the SDL system to the requirements of test generation and TTCN Link.
2. Generate a Link executable for the SDL specification.

Adapting the SDL System

The major adaptation of the SDL system that you have to do, is to modify it to properly describe the test architecture that is to be used. Basically, the requirement is that the channels from/to the environment of the system must correspond to the points of control and observation in the test suite. For example, the SDL system might be a specification of a communication protocol where the lower side of the protocol in practise can only be accessed through a network. To be able to create correct test cases, you also have to include the communication media in the SDL specification. Note however, that the specification of the media does not have to be a detailed specification of the functionality, only a specification of the aspects relevant to the current testing situation.

Generating a Link Executable

Once the SDL specification describing the system to be tested and the test architecture are finished, you can generate a Link executable. (A Link executable is sometimes also referred to as state space generator.) You do this in the same way as when you generate an SDL Simulator or Explorer.

To generate a Link executable:

1. Select *Make* from the *Generate* menu in the Organizer.

   The *Make* dialog will be opened.

2. Select *Analyze & generate code*.
3. Select *Generate Makefile*.
4. Select *Use Standard Kernel* and *TTCN Link*.
5. If necessary, change other options.
6. Click *Full Make*.

   The Link executable will now be generated. It includes the information about the SDL specification that is needed for generation of the TTCN declarations. The name of the executable will be `<sdl system name>_xxx.link`, where `xxx` is depending on the compiler used.

Note:

You should not change the SDL system after you have generated the Link executable. Such changes will not affect the generated Link executable and therefore not affect the generation of declarations.

Generating the Declarations

There are two steps involved in generation of the declarations (and the default table):

1. You select the Link Executable.
2. You start the generation.

Specifying the Link Executable

Before the actual generation of the TTCN declarations, you have to specify the Link executable – and thereby the SDL system – to use. There are two methods for doing this: associating the SDL and TTCN systems in the Organizer and explicitly selecting the Link executable in the TTCN Suite. In case a Link executable has been specified both in the Organizer and in the TTCN Suite, the one selected in the TTCN Suite is the executable that will be used.

Also note that the TTCN test suite that you are going to generate the declarations in, have to be created and added to the Organizer (and the same system file as the SDL system is included in).

Associating the SDL System with the TTCN Test Suite in the Organizer

1. In the Organizer, select the SDL system (the top node).
2. Select *Associate* from the *Edit* menu.

   The *Associate* dialog will be opened.
3. In the dialog, select the TTCN test suite and click *OK*.

   The association will be indicated by a new icon placed under the test suite icon.

Selecting the Link Executable in the TTCN Suite

1. Make sure that the test suite is opened and that the Browser is active.
2. From the *SDT Link* menu point to *Select Link Executable*. **On UNIX**, it is the menu in the Browser.

   The *Select Link Executable* dialog will be opened.
3. In the dialog, select the Link executable and click *OK*.

Note: External synonyms

The SDL system from which the Link executable is generated may contain external synonyms that do not have a corresponding macro definition (see "External Synonyms" on page 2650 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*). Such an SDL system cannot be used with TTCN Link and you will get an error message when trying to select the Link executable.

However, if you set the environment variable `SDTEXTSYNFILE` to a synonym definition file before starting SDL Suite or TTCN Suite, this file will automatically be used to define the external synonyms. If `SDTEXTSYNFILE` is set to "[[" all synonyms are given "null" values.
The syntax of a synonym file is described in "Reading Values at Program Start up" on page 2651 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*.

Generating the TTCN Declarations

When you have specified the Link executable, you can generate the declarations in the TTCN Suite:

1. Select *Generate Declarations* from the *SDT Link* menu. **On UNIX**, it is the menu in the Browser.

   This will generate the declarations and a default table.

2. Expand the Declarations Part and take a look:

   – One or more PCO type declarations have been generated.

   – For each channel to/from the environment in the SDL system, one PCO declaration has been generated.

   – For each signal on these channels, one ASN.1 ASP/PDU declaration has been generated.

   – For each data type that is used as a parameter on the signals to/from environment, a TTCN/ASN.1 data type definition has been generated if the data type cannot be mapped to a standard TTCN data type.

3. Expand the Dynamic Part. You should see that a default behavior tree called *Otherwise Fail* is generated. This contains otherwise statements with verdict FAIL for all generated PCOs.

PCO Mapping

There are two alternatives available for generation of the PCO types: either one PCO type is generated for each channel in the SDL system or only one PCO type is generated. This is defined by the configuration command `define-pco-type-mapping` (see "PCO Type Generation Strategy" on page 1406). The default is that only one PCO type is generated.

If more than one PCO type is generated, they are named *<Channel-Name>_TypeId*. If only one PCO type is generated it is called *PCO_Type*.

ASP/PDU Mapping

The generated ASPs/PDUs are given the same name as the corresponding SDL signal with one exception: If there are multiple PCO types and there is one signal that can be transported on more than one channel to the environment, this signal is divided into two ASPs, since an ASP may only be associated with one PCO type. The ASPs are then given names like *<SDLSignalName>_<PCOName>*.

By default, ASPs are generated from the SDL signals. However, you can change it to PDUs by using the `define-signal-mapping` command (see "SDL Signal Mapping Strategy" on page 1407).

Data Type Mapping

The data type mapping from SDL to TTCN/ASN.1 is defined in the following table. In most cases a table containing an ASN.1 type definition is generated for each data type. In the table below, this is indicated by a "<TTCN name> -> <ASN.1 definition>" clause. The <TTCN name> is the name used to denote the type in the test suite and the <ASN.1 definition> is the ASN.1 type definition that is the contents of the generated table. If the TTCN name is omitted, the name is given by the name of the corresponding SDL data type.

SDL data type	TTCN/ASN.1 data type
structure	-> SEQUENCE
array (with finite index sort)	-> SEQUENCE OF
string	-> SEQUENCE OF
bag	-> SET OF
enumerated type	-> ENUMERATED
boolean	BOOLEAN
character	Character -> IA5String (SIZE (1))
charstring	CharString -> IA5String
integer	INTEGER
real	Real -> REAL
natural	Natural -> INTEGER (0 .. MAX)

SDL data type	TTCN/ASN.1 data type
syntype	-> subtype
choice	-> CHOICE
bit	Bit -> BIT STRING (SIZE (1))
bit_string	BIT_STRING -> BIT STRING
octet	Octet -> OCTET STRING (SIZE (1))
octet_string	OCTET_STRING -> OCTET STRING
ObjectIdentifier	OBJECT_IDENTIFIER -> OBJECT IDENTIFIER
IA5String	IA5String
NumericString	NumericString
PrintableString	PrintableString
VisibleString	VisibleString
Null	NULL

In addition to the data types above, data types defined in external ASN.1 modules can also be used. These data types are mapped to definitions in the table named “ASN.1 Type Definitions by Reference” in the test suite. For each data type in the external ASN.1 module that is used on signals to/from the environment, one line defining the data type will be generated in this table. Note that the *Generate Declarations* command in the *SDT Link* menu assumes that the external ASN.1 module is setup as a dependency of the TTCN document.

No other data types than the ones mentioned above may occur on signals to/from the environment in the system.

Note that SDL is not case sensitive whereas TTCN is case sensitive. The spelling of the names generated by TTCN Link is given by the defining occurrence of the corresponding SDL name. Also note that no transformation of names is performed during generation of the TTCN names. This may in some cases lead to incorrect TTCN names if for example a reserved word from TTCN is used in the SDL system. To fix this problem, you have to change the name in the SDL system to a legal TTCN name.

Note that the SDL character NUL is mapped to NUL. Unfortunately, NUL is not an IA5String allowed character. So this must manually be changed to a legal character, e.g. ““. The NUL character is especially interesting since un-initialized SDL characters are set to NUL.

Modifying the Generated Declarations

In some cases, you may find it useful to manually modify the declarations that have been generated by TTCN Link before continuing with the development of the test cases. There are in particular two interesting cases:

- ASPs vs. PDUs.

  TTCN Link automatically generates ASPs for all signals visible on the border of the SDL system unless defined otherwise by the define-signal-mapping option (see “SDL Signal Mapping Strategy” on page 1407). If PDU definitions are more suitable for some of the signals, this is the time to change them. The simplest way is to copy the generated ASPs from the section *ASN.1 ASP Type Definitions* and paste them as PDUs in the section *ASN.1 PDU Type Definitions*.

- ASP field names.

  The ASPs are generated based on the SDL signals, and since the signals in SDL have no parameter names (only types), TTCN Link automatically generates names for the ASP fields. The fields are given the name “<type name><no>” where the <type name> is the name of the type of this parameter (but always starting with a non-capital letter to follow ASN.1 rules). It is however possible to change these names in the generated definitions, and if you do it before the test cases are developed, the new manually defined names will also be used in the test cases.

Regeneration of Declarations

It is possible to regenerate the declarations from an SDL system to incorporate new signals, channels and/or data type into the test suite. If you select *Generate Declarations* from the *SDT Link* menu again, only declarations with a name different from the existing test suite declarations will be inserted into the test suite.

Creating Test Cases

To create test cases with TTCN Link, you use the Table Editor in the TTCN Suite. When TTCN Link is used, the test cases are *synchronized*, that is, verified against the SDL specification.

In synchronized mode, the test case is guaranteed to be consistent with the specification. Each action you perform during the development of the test case will be incrementally verified by the state space exploration part of TTCN Link. When a table is in synchronized mode, the *SDT Link* menu of the **UNIX** Table Editor will contain new menu choices for editing of the test case. **In Windows**, you can find the corresponding commands in the *Link* dialog:

- *Send* will add a send statement to the test case. This is a manual step where you define the constraint to be associated with the send statement.
- *Receive* automatically generates all valid responses from the system under test. This implies that you do not need to check with the specification which possible signals the system can send in the state it is driven to by the proceeding lines in the test case.
- *Start timer* and *Cancel timer* are also manual commands where TTCN timers are started and cancelled. However, note that the timeout event corresponding to the timer will be automatically generated as a result of a *Receive* command.
- *Attach test step* will attach a previously defined test step, while still keeping the editor in synchronized mode.

If you modify the contents of the test case by using other menu choices, the editor will leave the synchronized mode.

The verdict for the generated test case lines, will always be either PASS or INCONCLUSIVE since the generated receive lines will always correspond to valid behaviors of the implementation under test.

A default test step, which consists of an otherwise fail for each PCO, will ensure that an incorrect response from the implementation under test always will give a FAIL verdict as a result from the test case.

Constraint Restrictions

The constraints that are used in send and receive statements in the test cases, are subject to certain restrictions:

- They may not use test suite or test case variables or test suite parameters.
- They must be “exactly” defined without any omits, any values, ranges, wildcards, etc.

The following is however allowed in send and receive constraints a test case is resynchronized, even though it is not generated automatically for receive constraints:

- The constraints can be structured/chained, that is, the constraints can reference and use other constraints defined in the test suite.
- The constraints can use test suite constants.
- The constraints can be parametrized.

Creating Test Steps

It is often useful to structure the test case into test steps. To do this by using TTCN Link:

1. Create the TTCN statements that should be in the test step directly in the test case table. This should of course be done in synchronized mode.
2. Cut the lines that should form the test step from the test case.
3. Create a test step table.
4. Paste the lines into the new test step table and adjust the indentation level.
5. Add an attach statement to the test case.

Showing SDL System Information

When you use TTCN Link for creating a test case, it is possible to access the SDL specification from the Table Editor.

To do this you first select a line in the test case. Then you have three alternatives:

- Select *Show SDL* from the *SDT Lin*k menu.

  The SDL Editor will be opened and display the executed SDL symbols that correspond to the selected test case line.

- Select *Show Coverage* from the *SDT Link* menu.

  This will display the Coverage Viewer and coverage information for the current test case.

- Select *Show MSC* from the *SDT Link* menu.

  The MSC Editor will be opened with a generated MSC diagram showing the execution path from the start of the SDL system to the state corresponding to the selected line in the test case. This may be particularly useful if you need to find out how unexpected receive statements are possible.

Merging TTCN Test Suites in the TTCN Suite

By using TTCN Link, you can only generate the declarations and create the dynamic tables. Either you could add the constraints and dynamic tables manually or merge the TTCN Link generated test suite with one generated by Autolink. A test suite generated by Autolink is in TTCN-MP format and contains constraints, declarations and dynamic tables.

To merge the test suites:

1. Make sure that the test suite that you want to merge the MP file into – that is, the *destination document* – is opened and active.

2. **In Windows**, select *Autolink Merge* from the *File* menu.

   On UNIX, select *Autolink Merge* from the *SDT Link* menu in the Browser.

   A dialog will be opened.

3. Find and select the MP file that you want to merge with the currently opened test suite.

4. Click *OK* **(in Windows)** or *Merge* **(on UNIX)**.

   The MP file you selected will be merged into the currently opened test suite.

   To complete the test suite **on UNIX**, you also have to generate the test suite overview. **In Windows**, the overview is generated automatically, for example before you print, and after that it is kept updated.

The merge will only work if the two test suites do not conflict. A conflict occurs if any TTCN object in the MP file has the same name as any TTCN object in the destination document. However, if such a conflict is detected, the merge will continue but the conflicting object in the MP file will be skipped.

Constraints will be merged in a special way. For example, the MP file may contain a TTCN ASP constraint called *constraint1* that refers to the type *type1* which is of the incompatible type TTCN PDU TypeDef. Because of this, a copy of *constraint1* will be inserted as a TTCN PDU constraint instead. However, this “type conversion” is limited. An ASN.1 constraint will not be converted to a TTCN constraint or vice versa.

Summary of TTCN Link

TTCN Link supports test case development and there are two major objectives:

- To help solving the consistency problem that arises as soon as there are two different descriptions of the same system, in this case the SDL specification and the TTCN test suite.
- To supply an environment that, based on the SDL specification, supports test suite development during the TTCN design. Both by directly using the SDL specification, for example to generate the declarations, and by providing access to the SDL specification directly from the TTCN Suite.

Overview of the TTCN Link Algorithm

This section will give a brief introduction to the algorithm used by TTCN Link to synchronize a test case with an SDL system.

The Composed System

The system that TTCN Link analyzes is the composed system that consists of both the SDL specification and the TTCN test case that is interactively created.

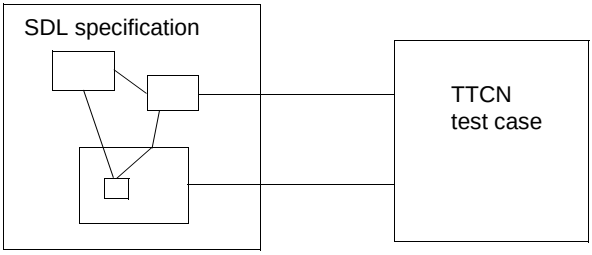


Figure 237: A composed SDL/TTCN system

The connection between the SDL system and the TTCN test case is created by connection of the channels to/from environment in the SDL system to the PCOs in the test suite.

State Space Exploration

The technique used by TTCN Link is based on state space exploration (sometimes referred to as reachability analysis) of the system composed of the SDL system together with the test case that is being created. The state space of this system can be viewed as a graph, where the nodes represent system states and the edges represent actions that can be performed by the system.

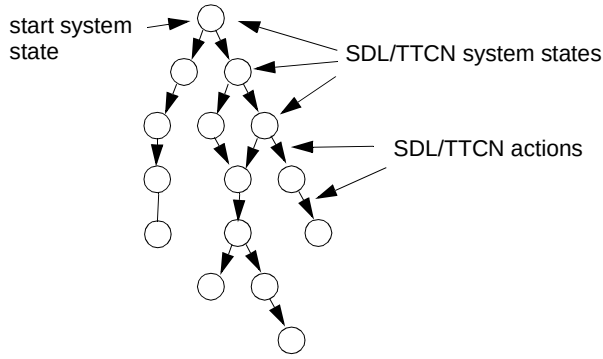


Figure 238: A state space fragment

Each system state represents the combined system in one moment in time. It contains information of for example:

- What SDL process instances exist
- The variable values of all the process instances
- The control flow state of the instances
- Any procedure calls, including local variables in the procedures
- The current line number of the test case
- Any started timers, both SDL and TTCN timers

The actions represented by the edges are either SDL actions like input, output, tasks, etc., or TTCN actions like send, receive or start timer.

Essentially, the algorithm to generate the state space of the combined system is the following, where two *global variables* – StateSpace (a graph that will contain the state space of the system) and TreatList (a list of states that is yet to be treated by the algorithm) – are used:

1. Create the start system state and add it to StateSpace and TreatList.
2. Remove one state (in step 3–4 called the current state) from TreatList.
3. Compute all possible actions that can be performed in the current state and the resulting system state that will be reached when the respective action has been performed.

4. For each action/resulting state:
   - If the resulting state was not already in StateSpace, add it to TreatList.
   - Add the action/resulting state to StateSpace.
5. If TreatList is empty: Terminate algorithm, the state space of the system is now represented by the graph in StateSpace.

   If TreatList is not empty: Go to step 2.

Incremental State Space Exploration

Since you interactively create the test case that describes the TTCN part of the combined SDL/TTCN system, it is not possible for TTCN Link to compute the entire state space at once. Instead, the state space exploration is performed in an incremental fashion in the following way:

1. You compute the state space that can be reached without any action by the test case.
2. When you TTCN Link to add a TTCN statement to a leaf in the test tree, you add the corresponding TTCN action(s)/resulting system state(s) to the state space.
3. Generate the state space that can be reached from the newly created system state(s) without any further action by the test case.
4. Go to step 2.

The consequence of this algorithm is that the state space of the combined SDL/TTCN system is explored in an incremental fashion, where each increment corresponds to a command you have given. Also the structure of the state space is influenced by the incremental way that it is generated. The state space can be visualized as a tree structure, where each node represents one line in the test case and a subpart of the state space, to be more precise, the subpart of the state space where the test case has executed this particular line but not the next one.

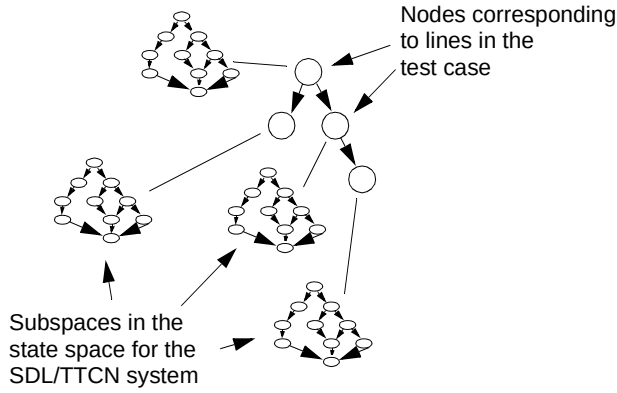


Figure 239: A structured state space

Since each line in the test case is created by a command from you, each node with its associated part of the state space can also be viewed as the state space increment that was created by a specific command.

Random Walk Exploration

The default state space exploration algorithm used by TTCN Link is the algorithm described in the previous sections. This is usually referred to as *exhaustive exploration* since it exhaustively explores the state space until all reachable states has been generated. The benefit of this algorithm is that when the exploration is finished, you can be sure that all possible combinations and alternatives are explored, that is, if TTCN Link generates two alternatives in a receive statement, the algorithm guarantees that there are no more valid alternatives. However, the drawback is that the algorithm requires all states in the state space to be kept in primary memory. For large SDL systems this may not always be possible with the computers available. The response time may also be unacceptable for interactive work with large systems.

To be able to use TTCN Link even in these cases, a second exploration algorithm is also provided. This is the *random walk* algorithm that, instead of exploring the entire state space, explores random paths in the state space using the algorithm described below. The input to the algorithm is a set of start states called StartList and a maximum depth of the exploration (MaxDepth) and the number of repetitions (Rep):

1. Select one state (in step 2–4 called the current state) from StartList.
2. Compute all possible actions that can be performed in the current state and the resulting system states that will be reached when the respective action has been performed.
3. If no actions could be performed from the current state or if the depth of the current random walk is MaxDepth, the current random walk is pruned. If the number of random walks performed so far is less than Rep, go to step 1, otherwise terminate the algorithm.
4. If actions could be performed and the depth is less than MaxDepth, select one of the generated states as a new current state and go to step 2.

The benefit with the random walk algorithm is that not more than a few system states (the current state and its successors) need to be kept in the memory at the time. The drawback is that there is no guarantee that the entire state space is explored, so, for example, even if TTCN Link generates only one receive alternative, it is possible that there are more alternatives.

To accomplish the best, both from exhaustive exploration and random walk, a two-step approach can be used when you use TTCN Link for large SDL systems:

1. Develop the test cases interactively by using the random walk algorithm with a small number of repetitions (1–3).
2. Verify that no more receive alternatives were possible by resynchronizing the test cases and using the exhaustive exploration algorithm. Or, if this is not possible due to lack of memory, use random walk with a high number of repetitions (at least 10, preferably 50–100).

This strategy gives both good performance when you interactively create the test cases and a good verification of the correctness of the test case during the automatic resynchronize. The execution time of the random walk algorithm is proportional to the number of repetitions.

You select which algorithm to use in the `.linkinit` file **(on UNIX)** or the `linkinit.com` file **(in Windows)** by using the `define-algorithm` command as described in section "Configuring the TTCN Link Executable" on page 1405.

Summary of the TTCN Link Algorithm

The algorithm used by TTCN Link to resynchronize a TTCN test case with an SDL system, is based on an incremental state space exploration of the state space of the composed system. The system consists of the SDL specification, together with the test case under construction. In the state space exploration, each command that you give will cause a new part of the state space to be explored, that is, the part that corresponds to the TTCN statement line that is inserted by the command. Two different exploration algorithms are available: exhaustive exploration and random walk. Exhaustive exploration is used for interactive development of test cases for a small SDL system and for verification of test cases for large systems. Random walk is used for interactive development of test cases for large SDL systems.

Configuring the TTCN Link Executable

This section describes the various options that can be used to configure the Link executable. You can change the options by giving the corresponding commands in a file called `.linkinit` **(on UNIX)** or `linkinit.com` **(in Windows)** in the target directory. The options that can be set are:

- Exploration Algorithm
- Random Walk Depth
- Random Walk Repetitions
- PCO Type Generation Strategy
- SDL Signal Mapping Strategy
- Stable State
- Timer Mode
- Transition
- Scheduling
- MSC Trace

These options will be described below.

It is also possible to add user-defined rules to the `.linkinit` file **(on UNIX)** or the `linkinit.com` file **(in Windows)** in order to prune the state space that is explored by Link. The user-defined rules are described in “User-Defined Rules” on page 1414.

Exploration Algorithm

What exploration algorithm is used, is defined by the command

```
define-algorithm [exhaustive|randomwalk]
```

and has the default value `exhaustive`. The differences between the different algorithms are described in section "Overview of the TTCN Link Algorithm" on page 1400.

Random Walk Depth

The maximum depth of each random walk is defined by the command

```
define-randomwalk-depth <integer>
```

and has the default value 500.

Random Walk Repetitions

The number of times a random walk is performed when a state space is explored by using this algorithm is defined by the command

```
define-randomwalk-repetitions <integer>
```

with a default value of 3.

PCO Type Generation Strategy

The PCO Type generation strategy is defined by the command

```
define-pco-type-mapping [system|channel]
```

and has default value `system`.

- If the parameter given is `system` then only one PCO type is generated for the entire system.
- If the parameter is `channel` then one PCO type is generated for each channel to/from the environment.

The choice also has an impact on the ASPs that are generated from SDL signals. Usually the name of the ASP is the same as the name of the SDL signal. However, if one PCO type is generated for each channel to/from the environment and one SDL signal can appear on more than one of these channels, then one ASP is generated for each channel it appears on. This is needed since an ASP can only be associated with one PCO type. The names of the signals are in this case defined as <signal name>_<channel name>.

SDL Signal Mapping Strategy

The SDL signals that appear on channels to/from the environment in the SDL system are either mapped to ASN.1 PDU or ASN.1 ASP definitions in the test suite. The mapping is defined by the command

```
define-signal-mapping [asp|pdu]
```

and has default value `asp`.

Stable State

The stable state option is defined by the command

```
define-stable [on|off]
```

and has the default value `on`.

It works like this:

Consider the situation when an empty test case has been resynchronized. The Link executable will now have computed the state space that can be reached without any input from or output to the tester. Usually, the state space looks something like this:

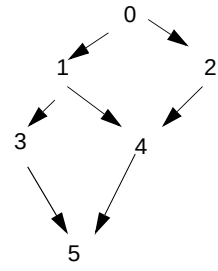


Figure 240: Original state space

where 0 is the start state, 1–4 are intermediate states and 5 is a stable state where all internal queues in the SDL system are empty and nothing more can happen without any input from the tester.

Let us now give a send command. This implies that new states are created as `send transitions` are added to the state space.

If the stable state assumption is `off` then we add one `send transition` to each state in the original state space, the new state space looks something like this:

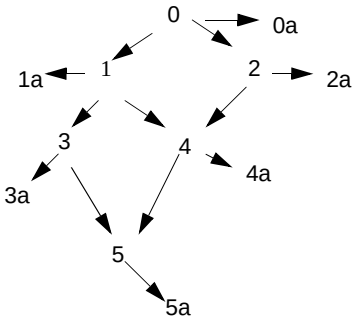


Figure 241: The new state space with stable state assumption "off"

where all states called something with 'a' are new. Now the states space that contains states that can be reached from the 'a' states without any input from or output to the tester is explored.

On the other hand, if the stable state assumption is `on` then we only add a `send transition` to the stable state, giving a new state space looking like:

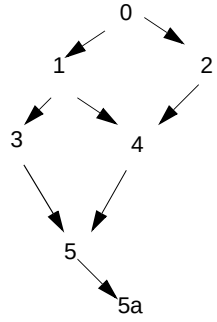


Figure 242: The new state space with stable state assumption "on"

Now, only the states that can be reached from "5a" without any input from or output to the tester are explored. This state space is of course a lot smaller that the one above.

Timer Mode

This is an option that in most cases will not have to be changed. The option defines how to interpret timeout actions compared with all other actions in the system. The option is changed by the command:

```
define-timer-mode [long|short]
```

The default is `long`.

If the timer mode is `long`, timeout actions will never occur if there is an internal event possible in the system. Essentially, the assumption is that the performance of the test system and IUT is good enough to ensure that the execution time for the actions are very small compared to the timeout times. Consider a system with the following state space when the `long` timer mode is used:

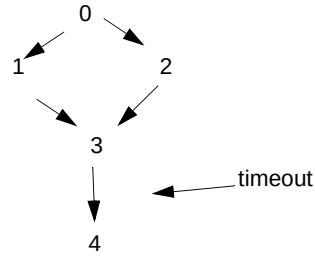


Figure 243: State space with timer mode "long"

In this system, the timeout event does not occur until no other event can happen. The transition leading to states 1–3 are all usual transitions – for example. inputs and outputs – so the timeout will only occur in state 3, where no other event is possible.

If the timer mode would have been `short`, the state space would have looked differently:

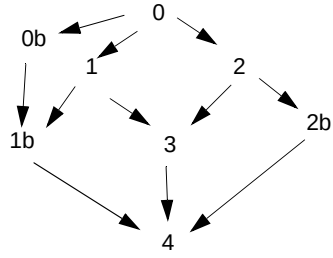


Figure 244: State space with timer mode "short"

The reason is that the timeout event now is possible in all the states 0–3.

Transition

The transition option defines what is considered to be an atomic transition in the state space and is defined by the command:

```
define-transition ['sdl'|'symbol']
```

The default value is `symbol`.

If the transition option is set to `sdl`, then SDL process graph transitions are considered atomic. This means that there will be no states in the state space where SDL processes are in the middle of a process graph transition. In all system states in the state space, the processes will always be in a process graph state.

If the transition option is set to `symbol`, the SDL process graph transitions are **not** considered to be atomic. In theory, this would imply that the processes could be interrupted anywhere during the execution of a transition. However, it turns out that for test generation purposes it is enough if the process graph transitions are divided at the points where one process communicates with another process, for example after an output or a create. A sequence of tasks or decisions is still viewed as atomic.

The consequence of this is that there will be more transitions in the state space if the transition option is `symbol` than if it is `sdl`. Consider a simple system with only one process. Let this process have a transition like:

```
state xx;
  input st1;
    output sig2;
    output sig3;
    output sig4
    nextstate  st2;
```

If the transition option is `sdl`, there will only be one transition in the state space that corresponds to the process graph transition above:

```
X : process is in state st1
|
Y : process is in state st2
```

If the transition option is `symbol`, there will be a sequence of transitions in the state space:

```
X : process is in state st1
| input st1; output sig2;
X1
| output sig2;
X2
| output sig3;
X3
| output sig4; nextstate  st2;
Y : process is in state st2
```

This will of course give a lot bigger state space.

Scheduling

The scheduling option is defined by the command

```
define-scheduling [‘first’|’all’]
```

This option controls how many processes are allowed to execute in a given system state. If the option is set to `first`, only one process (the first in the ready queue) is allowed to execute. If the option is set to `all`, all processes that can execute are allowed to do it. The default value is `all`.

Consider an SDL system with two static processes. If the scheduling option is `all`, the initial part of the state space for this system will probably look like:

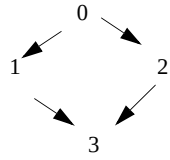


Figure 245: State space with scheduling as "all"

where

- 0 is the initial state (both processes are in the `start symbol`).
- 1 is the state where the first process has executed its start transition while the second process is still in its `start symbol`.
- 2 is the state where the second process has executed its start transition while the first process is still in its `start symbol`.
- 3 is the state where both processes have executed their start transitions.

On the other hand, if the scheduling option is `first`, it will look like:

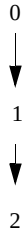


Figure 246: State space with scheduling as "first"

where

- 0 is the initial state (both processes are in the `start symbol`).
- 1 is the state where the first process has executed its start transition while the second process is still in its `start symbol`.
- 2 is the state where both processes have executed their start transitions.

If there are lots of processes, there will be a significant difference in the size of the state space depending on how the `scheduling` option is set!

MSC Trace

The MSC trace options control what is displayed in the MSCs generated by TTCN Link. There are two different MSC trace options controlling if states and actions in the SDL system are showed as MSC condition symbols and MSC action symbols. The options are set by the following commands:

```
define-MSC-trace-actions [ 'on' | 'off' ]

define-MSC-trace-states [ 'on' | 'off' ]
```

The default value for both options is `'off'`.

An Example of a .linkinit / linkinit.com File

This following `.linkinit` file **(on UNIX)** or `linkinit.com` file **(in Windows)** will change the exploration algorithm to random walk and set the number of repetitions to 2:

```
define-algorithm random
define-random-rep 2
```

This is a useful configuration when you work interactively with TTCN Link, since the random walk algorithm is quicker and requires less memory than the exhaustive algorithm. But, since the random walk in some cases can miss some alternative receive statement, a useful strategy is the following: Use the configuration above when you work interactively with TTCN Link. However, when you have finished with a test case or a number of test cases, check that the assumptions are valid by resynchronizing with the exhaustive algorithm or a larger number of repetitions.

Note that in the commands in the `.linkinit` file **(on UNIX)** and the `linkinit.com` file **(in Windows)** can be abbreviated as long as they are unique.

User-Defined Rules

User-defined rules can be used during state space exploration to prune the search performed by the TTCN Link. Whenever a system state is found that matches the defined rule, the search is pruned at this particular state. This can be useful in order to remove specific exceptional behavior from the test cases that are designed and instead handle these in a special default test step.

Consider an SDL specification that contains a `clock` process that has a timer `intervaltimeout`. Every time the `intervaltimeout` expires, a signal `DisplayTime` is sent to the environment of the SDL system. Since this can happen at any time during the execution, there will be an alternative at all receive statements corresponding to the reception of this signal. To avoid this, it is better to add a receive statement in the default dynamic behaviour that skips this signal.

To achieve this with TTCN Link, you have to do two things:

- Define a rule that prunes the state space at appropriate places.
- Modify the default dynamic behaviour that is generated by TTCN Link.

A rule that prunes the state space whenever the `intervaltimer` expires is the following:

```
def-rule sitype(signal(clock:1))=intervaltimeout;
```

The statements that need to be added to the default behaviour are:

```
PCO?DisplayTime            DisplayTime_Match_All
          RETURN
```

These lines will cause the tester to ignore DisplayTime signals sent from the system.

A rule essentially gives the possibility to define predicates which describe properties of one particular system state. As soon as this predicate matches a system state, TTCN Link will prune the search. A rule consists of a predicate (as described below) followed by a semicolon (';'). In a rule, all identifiers and reserved words can be abbreviated as long as they are unique.

Note:

Only one rule can be used at any moment. If more than one rule is needed, reformulate the rules as one rule, by using the boolean operators described below.

Predicates

The following types of predicates exist:

- Quantifiers over process instances and signals in input ports
- Boolean operator predicates such as “and”, “not” and “or”
- Relational operator predicates such as “=” and “>”

Parenthesis are allowed to group predicates.

Quantifiers

The quantifiers listed below are used to define rule variables denoting process instances or signals. The rule variables can be used in process or signal functions described later in this section.

```
exists <RULE VARIABLE> [: <PROCESS TYPE>]
[ | <PREDICATE>]
```

This predicate is true if there exists a process instance (of the specified type) for which the specified predicate is true. Both the process type and the predicate can be excluded. If the process type is excluded, all process instances are checked. If the predicate is excluded, it is considered to be true.

```
all <RULE VARIABLE> [ : <PROCESS TYPE>]
[ | <PREDICATE>]
```

This predicate is true for all process instances (of the specified type) for which the specified predicate is true. Both the process type and the predicate can be excluded. If the process type is excluded, all process instances are checked. If the predicate is excluded, it is considered to be true.

```
siexists <RULE VARIABLE> [ : <SIGNAL TYPE>]
[ -  <PROCESS INSTANCE>] [ |  <PREDICATE>]
```

This predicate is true if a signal (of the specified type) exists in the input port of the specified process for which the specified predicate is true. If no signal type is specified, all signals are considered. If no process instance is specified, the input ports of all process instances are consid-

ered. If no predicate is specified, it is considered to be true. The specified process can be either a rule variable that has previously been defined in an `exists` or `all` predicate, or a process instance identifier (`<PROCESS TYPE>:<INSTANCE NO>`).

```
siall <RULE VARIABLE> [ :  <SIGNAL TYPE>]
[ - <PROCESS INSTANCE>] [ | <PREDICATE>]
```

This predicate is true for all signals (of the specified type) in the input port of the specified process for which the specified predicate is true. If no signal type is specified, all signals are considered. If no process is specified, the input ports of all process instances are considered. If no predicate is specified, it is considered to be true. The specified process can be either a rule variable that has previously been defined in an `exists` or `all` predicate, or a process instance identifier (`<PROCESS TYPE>:<INSTANCE NO>`).

Boolean Operator Predicates

The following boolean operators are included (with the conventional interpretation):

```
not <PREDICATE>
<PREDICATE> and <PREDICATE>
<PREDICATE> or <PREDICATE>
```

The operators are listed in priority order, but the priority can be changed by parenthesis.

Relational Operator Predicates

The following relational operator predicates exist:

```
<EXPRESSION> = <EXPRESSION>
<EXPRESSION> != <EXPRESSION>
<EXPRESSION> < <EXPRESSION>
<EXPRESSION> > <EXPRESSION>
<EXPRESSION> <= <EXPRESSION>
<EXPRESSION> >= <EXPRESSION>
```

The interpretation of these predicates is conventional. The operators are only applicable to data types for which they are defined.

Expressions

The expressions that are possible to use in relational operator predicates are of the following categories:

- Process functions: Extract values from process instances

- Signal functions: Extract values from signals
- Global functions: Examine global aspects of the system state
- SDL literals: Conventional SDL constant values

Process Functions

Most of the process functions must have a process instance as a parameter. This process instance can be either a rule variable that has previously been defined in an `exists` or `all` predicate, a process instance identifier (`<PROCESS TYPE>:<INSTANCE NO>`) or a function that returns a process instance, e.g. `sender` or `from`.

```
state( <PROCESS INSTANCE> )
```

Returns the current SDL state of the process instance.

```
type( <PROCESS INSTANCE> )
```

Returns the type of the process instance.

```
iplen( <PROCESS INSTANCE> )
```

Returns the length of the input port queue of the process instance.

```
sender( <PROCESS INSTANCE> )
```

Returns the value of the imperative operator sender (a process instance) for the process instance.

```
parent( <PROCESS INSTANCE> )
```

Returns the value of the imperative operator parent (a process instance) for the process instance.

```
offspring( <PROCESS INSTANCE> )
```

Returns the value of the imperative operator offspring (a process instance) for the process instance.

```
self( <PROCESS INSTANCE> )
```

Returns the value of the imperative operator self (a process instance) for the process instance.

```
signal( <PROCESS INSTANCE> )
```

Returns the signal that is to be consumed if the process instance is in an SDL state. Otherwise, if the process instance is in the middle of an SDL process graph transition, it returns the signal that was consumed in the last input statement.

```
<PROCESS INSTANCE> -> <VARIABLE NAME>
```

Returns the value of the specified variable. If `<PROCESS INSTANCE>` is a previously defined rule variable, the `exists` or `all` predicate that defined the rule variable must also include a process type specification.

```
<RULE VARIABLE>
```

Returns the process instance value of `<RULE VARIABLE>`, which must be a rule variable bound to a process instance in an `exists` or `all` predicate.

Signal Functions

Most of the signal functions must have a signal as a parameter. This signal can be either a rule variable that has previously been defined in an `siexists` or `siall` predicate, or a function that returns a signal, e.g. `signal`.

```
sitype( <SIGNAL> )
```

Returns the type of the signal.

```
to( <SIGNAL> )
```

Returns the process instance value of the receiver of the signal.

```
from( <SIGNAL> )
```

Gives the process instance value of the sender of the signal.

```
<RULE VARIABLE> -> <PARAMETER NUMBER>
```

Returns the value of the specified signal parameter. The `siexists` or `siall` predicate that defined the rule variable must also include a signal type specification.

```
<RULE VARIABLE>
```

Returns the signal value of `<RULE VARIABLE>`, which must be a rule variable bound to a signal in a `siexists` or `siall` predicate.

Global Functions

```
maxlen( )
```

Gives the length of the longest input port queue in the system.

```
instno( [<PROCESS TYPE>] )
```

Returns the number of instances of type `<PROCESS TYPE>`. If `<PROCESS TYPE>` is excluded the total number of process instances is returned.

```
depth( )
```

Gives the depth of the current system state in the behavior tree/state space.

SDL Literals

```
<STATE ID>
```

The name of an SDL state.

```
<PROCESS TYPE>
```

The name of a process type.

```
<PROCESS INSTANCE>
```

A process instance identifier of the format `<PROCESS TYPE>:<INSTANCE NO>`, e.g. Initiator:1.

```
<SIGNAL TYPE>
```

The name of a signal type.

```
null
```

SDL null process instance value

```
env
```

Returns the value of the process instance in the environment that is the sender of all signals sent from the environment of the SDL system.

```
<INTEGER LITERAL>
true
false
<REAL LITERAL>
<CHARACTER LITERAL>
<CHARSTRING LITERAL>
```

SDL Restrictions

The restrictions imposed on the SDL specification by TTCN Link are basically of four different kinds:

- General SDL restrictions
- State space exploration restrictions
- Data type mapping restrictions
- TTCN name restrictions

General SDL Restrictions

TTCN Link is based on the SDL to C compiler and has thus the same restrictions as the SDL Simulator and Explorer which are also based on the SDL to C compiler. The major restrictions are:

- No context parameters
- No channel substructures
- No signal refinements
- No axioms, literal mappings, inheritance or name class literals in abstract data types

For more information about the general SDL restrictions see "SDL Restrictions" on page 7 in chapter 1, *Compatibility Notes, in the Release Guide*.

State Space Exploration Restrictions

TTCN Link is based on state space exploration of the combined state space of the SDL system and the TTCN test case. Since there is only a finite amount of memory available in computers, this means that there will be restrictions on the size of the state space that can be handled. It is not possible to give a numeric value on this restriction since it depends both on the SDL system and on the computer, but TTCN Link has been successfully used on SDL systems with more than 10 processes on a SPARCstation 10 computer. Also see "Overview of the TTCN Link Algorithm" on page 1400.

Data Type Mapping Restrictions

Only the data types described in section "Generating the Declarations" on page 1390 are allowed on the channels to/from the system.

TTCN Name Restrictions

The mapping of concepts from SDL to TTCN generates a lot of names in TTCN. For example, the signals in SDL will become ASPs/PDUs in TTCN and SDL data types will become TTCN data types. The names of the generated entities are taken directly from the names on the corresponding SDL entity. This will lead to problems if, for example, the names are reserved words in TTCN. In this case, the names in the SDL system have to be changed.

TTCN Link Commands in the TTCN Suite

TTCN Link Commands in the TTCN Suite on UNIX

This section describes the extra menu choices that are available in the TTCN Suite when TTCN Link is used **on UNIX**.

Browser Commands in the *SDT Link* Menu

The following menu choices are available in the Browser *SDT Link* menu:

- *Select Link Executable*
- *Generate Declarations*

Select Link Executable

Makes a connection between a test suite and the corresponding SDL system. In the file dialog that opens, a Link executable should be selected. If the file is not a legal Link executable, the selection will fail.

When a Link executable is selected, a place holder for it is stored in the test suite. It is possible to change the Link executable if, and only if, there is no test case which uses the current SDL system (i.e. there is no synchronized test case).

It is also possible to select a Link executable by associating the SDL system with the TTCN system in the Organizer. However, a Link executable selected in the TTCN Suite will override an executable selected in the Organizer.

See also "External synonyms" on page 1391.

Generate Declarations

Generates TTCN versions of the relevant type declarations in the SDL system. The menu choice is only available if a Link executable has been selected.

The generated objects use the ASN.1 syntax. They are automatically analyzed after they have been generated. This is necessary for later operations and usage of these types. If there is no other declarations (types) in the test suite (e.g. the test suite is empty), the analysis will not fail. On the contrary if other declarations (types) already exist, the analysis

may fail due to name conflicts and incorrect references. The error messages of this analyzing will not be displayed. To check if the generated declarations are analyzed, use the Selector and the *Show Error Message* command on the incorrect tables.

At the same time that the declarations are generated, a Default table will be generated. It consists of an otherwise statement for each PCO and a timeout statement.

No timer will be generated from the SDL system. If the design of the test suite requires any timers they must be defined manually.

More details about the generated tables etc. can be found in "Generating the Declarations" on page 1390.

Table Editor Commands in the *SDT Link* Menu

To generate a test case (the behaviour description of a test case), the test case table must be in synchronized mode. In synchronized mode, the test case is synchronized (has an established connection) with the selected Link executable.

Once in synchronized mode, the test case editor will stay in this mode as long as only commands from the *SDT Link* menu are used. As soon as any field in the table (besides the comment fields) is edited, the synchronized mode will be terminated. It is however possible to analyze the test case without leaving the synchronized mode.

The following commands are available from a Table Editor for test cases or test steps. They are applied on a behaviour line (they insert a new behaviour line below/after the behaviour line with the input focus) hence it is required that the test case or the test step either is empty or has the input focus on a leaf row.

When each of these commands is performed the input focus is moved to the new generated behaviour line.

- *Send*
- *Receive*
- *Start Timer*
- *Cancel Timer*
- *Attach*
- *Resynchronize*
- *Show SDL*
- *Show MSC*

- *Show Coverage*
- *Show Options*

Send

Adds a send statement below/after the behaviour line with the input focus. The new behaviour line will have an increased indent level compared with the previous one.

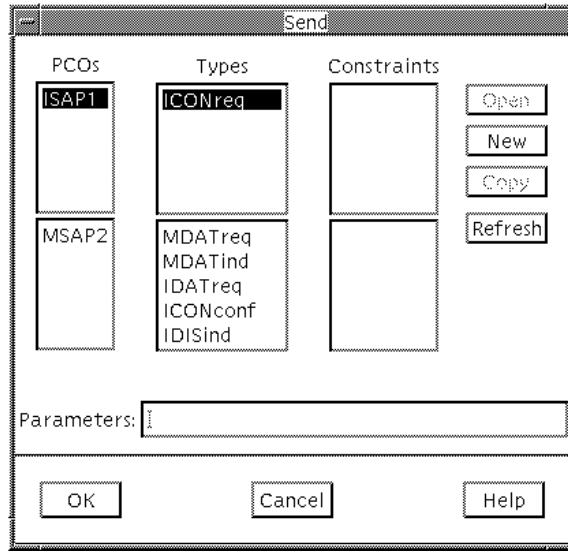


Figure 247: Send dialog

The send statement with the selected PCO, ASP/PDU and constraint will be verified by the selected SDL system. If the verification does not fail, a send statement is generated and inserted below/after the row with the input focus. For a more detailed description of this dialog, see "Add Send Statement" on page 1180 in chapter 25, *The TTCN Table Editor (on UNIX)*.

Receive

Adds appropriated receive and/or timeout statement(s) below/after the behaviour line with the input focus. The new row(s) will have the same

indent level. This indent level will be increased compared with the previous one.

For each retrieved receive statement from the SDL system, a new Constraint is generated if there is no appropriate Constraint (a Constraint with a similar value). The new Constraint will be named with a unique name and will be analyzed. This new name is used in the *Constraints Ref* column.

Start Timer

Adds a start timer statement below/after the behaviour line with the input focus. The new behaviour line will have an increased indent level compared to the previous one.

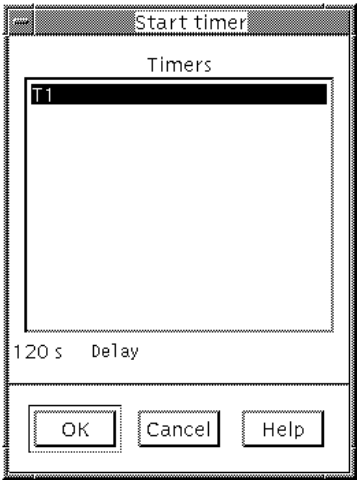


Figure 248: Start Timer dialog

The start timer statement with the selected timer will be verified with the selected SDL system. If the verification does not fail, a start timer statement is generated and inserted below/after the behaviour line with the input focus. For a more detailed description of this dialog see "Add Send Statement" on page 1180 in chapter 25, *The TTCN Table Editor (on UNIX)*.

Cancel Timer

Analogous to *Start Timer*.

Attach

Adds an attachment statement below/after the row with the input focus. The new behaviour line will have an increased indent level compared to the previous one.

The test step dialog used by this command is similar to the dialog in the Table Editor. The selected test step (the behaviour lines in the behaviour description) will be verified with the SDL system. The test step must have passed analysis before this operation.

Resynchronize

Verifies the test case in a Table Editor using the previously chosen Link executable. The table will change mode to the synchronized mode. This command is available from Table Editors for test cases and only if an SDL system is selected.

If the test case does not have any default reference and there is more than one default in the test suite, a selection dialog pops up and a default must be selected. If the test suite contains only one default, it will be selected automatically. If the test case already has a default, no change will be made.

If the test case (or the test step) contains behaviour lines, they will be verified with the current SDL system. If the verification of any line fails, the table will keep the normal mode.

Show SDL

Opens the SDL Editor with the symbols selected which were executed in the SDL system and are associated with the behaviour line which has the input focus. More precisely, the SDL symbols which were executed after the current test case line but before the next test case line, are selected exactly.

Show MSC

Opens the MSC Editor with a process level MSC that illustrates the execution path from the start of the SDL system to the state corresponding to the behaviour line with input focus.

Show Coverage

Opens the Coverage Viewer that displays test coverage information for the current test case. The test coverage displays how many times each symbol in the SDL system has been executed during the generation of the test case. Note that the important information is not the exact number of times a particular symbol has been executed (since this is dependent upon the particular algorithm used by the Link executable). The important information is whether a symbol has been executed or not. If a symbol in the SDL system has not been executed when generating the test case, the requirement defined by this symbol is not tested by the test case.

Show Options

Shows the current settings of the configuration parameters that control the way the Link executable explores the state space of the combined SDL/TTCN system.

TTCN Link Commands in the TTCN Suite in Windows

In Windows, different TTCN Link commands are included in the *SDT Link* menu and the *Link* dialog. The menu choices in the *SDT Link* menu are used for selecting the Link executable, generating declarations and for showing execution information. By using the *Link* dialog, you can generate various statements.

The *SDT Link* menu and the *Link* dialog will be explained below.

The *SDT Link* Menu

The following menu choices are included in the *SDT Link* menu:

- *Select Link Executable*
- *Generate Declarations*
- *Show SDL*
- *Show MSC*
- *Show Coverage*
- *Show Options*

Select Link Executable

Makes a connection between a test suite and the corresponding SDL system. Opens a dialog in which you may select the Link executable.

It is also possible to specify the Link executable in the Organizer by associating the SDL system with the TTCN system. However, a Link executable selected in the TTCN Suite will override an executable selected in the Organizer.

See also "External synonyms" on page 1391.

Generate Declarations

Generates TTCN versions of the relevant type declarations in the SDL system. The menu choice is only available if a Link executable has been selected.

The generated objects use the ASN.1 syntax. They are automatically analyzed after they have been generated. This is necessary for later operations and usage of these types.

At the same time as the declarations are generated, a Default table will be generated. It consists of an otherwise statement for each PCO and a timeout statement.

Timers will not be generated from the SDL system. If the design of the test suite requires any timers, they must be defined manually.

Show SDL

Opens an SDL Editor with the symbols selected which were executed in the SDL system and are associated with the selected behaviour line. More precisely, the SDL symbols which were executed after the current test case line but before the next test case line, are selected exactly.

Show MSC

Opens an MSC Editor with a process level MSC that illustrates the execution path from the start of the SDL system to the state corresponding to selected the behaviour line.

Show Coverage

Opens an SDL Coverage Viewer that displays test coverage information for the current test case. The test coverage displays how many times each symbol in the SDL system has been executed during the generation of the test case. Note that the important information is not the exact number of times a particular symbol has been executed (since this is dependent upon the particular algorithm used by the Link executable). The important information is whether a symbol has been executed or not. If a symbol in the SDL system has not been executed when the test case was generated, the requirement defined by this symbol is not tested by the test case.

Show Options

Shows the current settings of the configuration parameters that control the way the Link executable explores the state space of the combined SDL/TTCN system.

The *Link* Dialog

The *Link* dialog can be opened from the *SDT Link* menu. The *Link* dialog can only be used when a behaviour line is selected in the table or if the table does not yet contain a behaviour line. When the *Link* dialog is opened, the current table automatically becomes synchronized with the Link executable, that is, the table is read-only. The only time synchronization is lost for the current table, is when TTCN Link is activated in another table or when you insert a behaviour line from the *Data Dictionary* dialog in the current table.

The *Link* dialog has almost the same appearance as the *Data Dictionary* dialog. A list in the lower left corner of the dialog makes it possible to switch between the *Link* and *Data Dictionary* dialog. For more information about the Data Dictionary, see "Creating Behaviour Lines" on page 1269 in chapter 30, *Editing TTCN Documents (in Windows)*.

The operations available in the *Link* dialog will be described below. The operations are applied on a selected behaviour line and the result is that a new behaviour line is inserted below/after. The test case or the test step must be empty or a leaf row must be selected.

The *Send/Receive* Tab

Generate a send statement by selecting a PCO, ASP/PDU, constraint, etc. When you press the *Apply* button, the statement will be verified by the selected SDL system. If the verification succeeds, a new send statement will be inserted below the selected behaviour line.

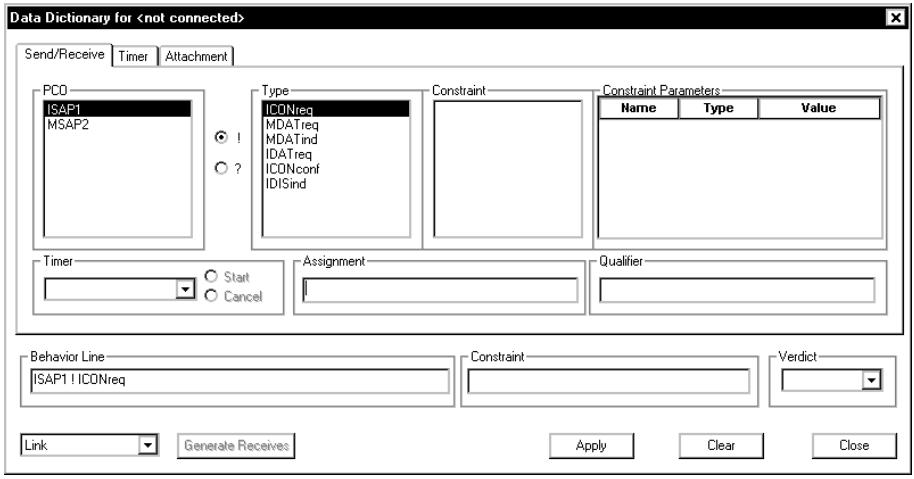


Figure 249: Generating send statements

Generate receives and/or timeout statement(s) below the selected behaviour line by clicking the *Generate Receives* button. The new row(s) will all have the same indent level. This indent level will be increased compared with the previous one.

This operation is only valid when the current selected behaviour line is a leaf row.

For each retrieved receive statement from the SDL system, a new Constraint is generated if there is no appropriate constraint (a constraint with a similar value). The new constraint will be named with a unique name and will be analyzed. This new name is used in the *Constraints Ref* column.

The *Timer* Tab

Generate a timer statement by selecting *Start* or *Cancel* and a timer from the listbox (optional for *Cancel*). When you click the *Apply* button, the timer statement will be verified against the SDL system. If the verification succeeds, a timer statement will be generated and inserted below the selected behaviour line. The new behaviour line will have an increased indent level compared to the previous one.

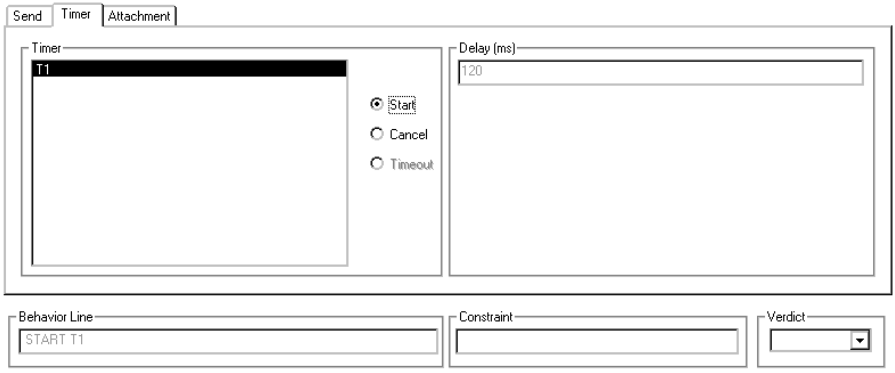


Figure 250: Generating timer statements

The *Attachment* Tab

When you select an attachment and click *Apply*, an attachment statement will be generated below the selected behaviour line. The new line will have an increased indent level compared to the previous one.

The selected test step (the behaviour lines in the behaviour description) will be verified with the SDL system. The test step must have passed analysis before this operation.

Note:

TTCN Link does not support attachment parameters.

Using Autolink

The generation of a TTCN test suite with Autolink proceeds in several steps.

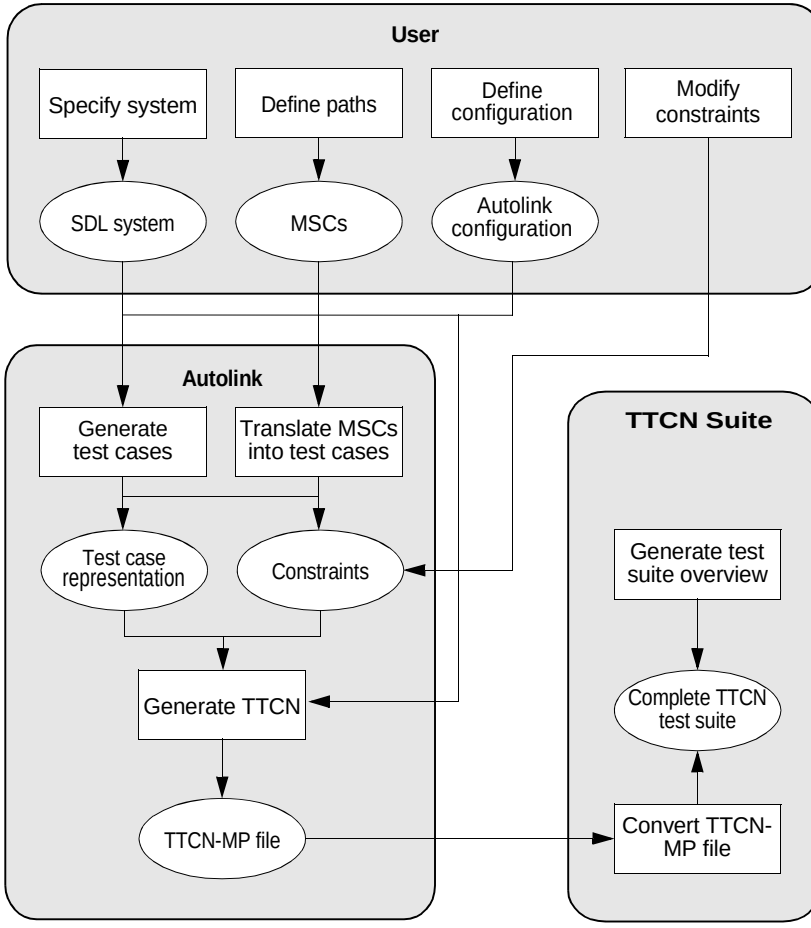


Figure 251: Test suite generation with Autolink

You start by specifying an SDL system, see "Specifying the SDL System and Performing Other Preparations" on page 1432. Based on this SDL specification, you generate an SDL Explorer application which includes Autolink.

Next, you define MSC test cases, see "Defining MSC Test Cases" on page 1434, which describe the purpose of the test cases of your test suite. They are stored on disk as *system level MSCs*, that is, MSCs with only one instance axis for the SDL system and one or more instance axis for the environment. Test cases may contain *test steps* which are stored as separate system level MSCs on disk.

You may also want to define an Autolink configuration, see "Defining an Autolink Configuration" on page 1449, in order to guide the naming and parameterization of constraints. Autolink can also be configured to store test cases into a hierarchy of test groups.

The next step is to generate test cases, see "Translating MSCs into Test Cases" on page 1454. This can be done either by a state space exploration of the SDL system or by directly translating system level MSCs into test cases. In any case, the result is an internal representation for each test case. At the same time, a list of constraints is generated. These constraints can be renamed and merged, see "Modifying Constraints" on page 1455. You can also add new constraints manually.

Finally, you generate a TTCN test suite, see "Generating a TTCN Test Suite" on page 1457, based on the test case representations and the list of constraints. The test suite is stored on disk in TTCN-MP format. **On UNIX**, you can import this file in the TTCN Suite; **in Windows**, you can simply open it in the TTCN Suite. On both platforms, you can convert the TTCN-MP file to the graphical TTCN format in the Organizer.

On the following pages, the steps will be described in more detail.

Specifying the SDL System and Performing Other Preparations

Before you start the SDL Explorer, directories must be created where you will store test case and test step representations. If there are no appropriate directories:

- Create one directory for test cases and one directory for test steps in your working directory.

Specifying the SDL System

You have to specify an SDL system in order to create an SDL Explorer. At minimum, you must specify all channels to the environment of your system and all signals sent via these channels. With such a minimal specification, you can use Autolink to translate MSCs directly into TTCN by using the Translate-MSC-Into-Test-Case command. The advantages and disadvantages of using this command are described in "Translating MSCs into Test Cases" on page 1454.

Generating and Starting an SDL Explorer

When you have specified the SDL system, you can generate and start the Explorer. How to do this is described in "Generating and Starting an SDL Explorer" on page 2393 in chapter 53, *Validating a System*.

Specifying Directories

Before you start defining test cases and test steps, you have to specify where they are to be saved:

1. In the *Explorer*, select *Autolink: Test Cases Directory* from the *Options2* menu.
   - This corresponds to the command Define-MSC-Test-Cases-Directory.
2. In the dialog that will be displayed, select the test case directory that you previously created and click *OK*.
3. Select *Autolink: Test Steps Directory* from the *Options2* menu.
   - This corresponds to the command Define-MSC-Test-Steps-Directory.
4. In the dialog that will be displayed, select the test step directory that you previously created and click *OK*.

When you later leave the Explorer, you can save these values.

Defining MSC Test Cases

In Autolink, an MSC test case is derived from a *path*. A path is a sequence of events that have to be performed in order to go from a start state to an end state. There are two ways to define MSC test cases:

1. Interactive simulation and manual specification
2. Automatic computation by Autolink (see "Defining MSC Test Cases Automatically - Coverage Based Test Generation" on page 1448)

Defining MSC Test Cases Interactively

The creation of MSC test cases by interactive simulation proceeds in several steps:

1. Specify the start state of the test case.

   If this state is identical to the root of the behavior tree, nothing has to be done. Otherwise, you must navigate to the desired state, for example by using the Navigator, selecting a previously defined report or verifying an MSC. Then you set the root to the current state with the `Define-Root Current` command.

> **Note:**
>
> Autolink always considers the current root of the behavior tree to be the start state of a path.
>
> Also note that when a test case is generated, the root has to be the same as it was at the moment of the test case definition. You have to keep track of the start state with Print-Path and Goto-Path, for example if you want to leave the Explorer temporarily.

2. Navigate through the system to the desired end state.
3. Select *MSC: Save Test Case* from the *Autolink1* menu.
   – This corresponds to the Save-MSC-Test-Case command.

An MSC test case consists of one instance axis for the SDL system and a separate instance axis for each channel to/from the environment. In TTCN terms, the single SDL system instance represents the *System Under Test (SUT)*. The environment instances represent the *Points of Control and Observation (PCO)*; PCOs are the interface between the *test system* and the SUT.

The system level MSC that will be saved contains the *observable events* of the path between the start and the end state. Observable events represent the external interaction that takes place between the SDL system and its environment. (During conformance testing, external interaction takes place between the implementation and the test system.)

Figure 252 shows an MSC test case.

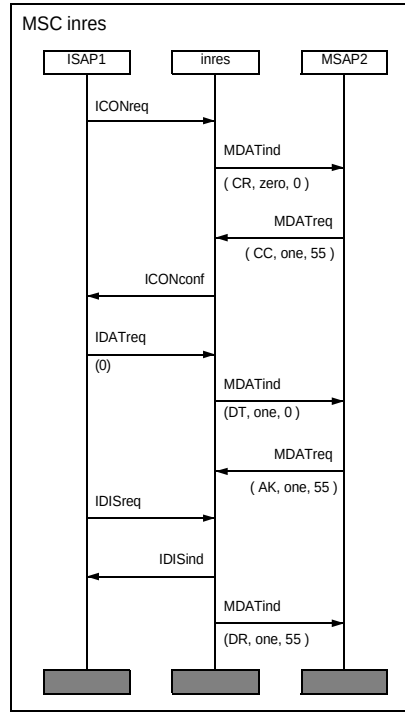


Figure 252: An MSC test case

Incorporating Test Steps in Test Cases

Typically, test cases are structured logically into several parts, for example a preamble, a test body and a postamble. These parts are called test steps. You may incorporate test steps in a test case by using MSC references.

Figure 253 shows an MSC with two MSC references. Each of the referenced MSCs represents a separate test step; they are called *Preamble* and *Postamble*.

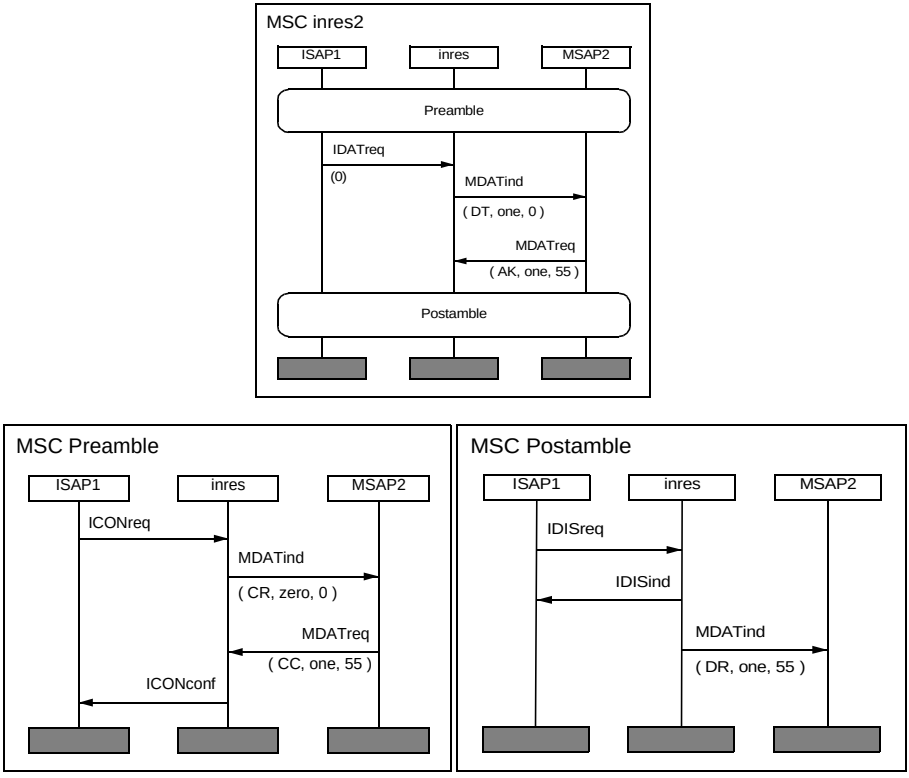


Figure 253: An MSC with a preamble and a postamble

Test steps are stored with the same file extension as test cases (`.mpr`). They are created in analogy to test cases with Save-MSC-Test-Step.

If you want to create a test case with a preamble and a postamble, several steps are necessary:

1. Make sure that you have specified the directories for test cases and test steps as described in "Specifying Directories" on page 1433.

2. Set the root of the state space to the start state of the test case by using the command Define-Root Current.
3. Navigate to the end of the path of the preamble.
4. Use Save-MSC-Test-Step to save the preamble.
5. Set the root of the state space to the current state by using the command Define-Root Current.
6. Navigate to the end of the path of the test body.
7. Use Save-MSC-Test-Case to save the test case/test body.
8. Set the root of the space to the current state.
9. Navigate to the end state of the test case.
10. Use Save-MSC-Test-Step to save the postamble.
11. Add two MSC references manually to the MSC test case with the MSC Editor.

Alternatively, you may create a single MSC test case and split the file into preamble, test body and postamble afterwards.

Test steps may refer to other test steps, but not to test cases. During test case generation, Autolink keeps track of the nested structure of test cases and test steps.

Note:

When Autolink generates test cases (see Generate-Test-Case and Translate-MSC-Into-Test-Case), the semantics of MSC references and MSC reference expressions are different from the semantics given in the ITU-T Recommendation Z.120!

Autolink requires that a test step is completely evaluated before the next test step starts, i.e. it synchronizes among references, whereas Z.120 considers MSC references as macros which do not have to be evaluated as a unit.

With regard to Figure 253, this means that all signals of the test body in `mi_inres2` have to be evaluated before the postamble starts.

Using Timers

Autolink supports test suite timers. There are three types of timers which are commonly needed in test sequences:

1. A global timer is specified to guarantee that test cases end even if they are blocked during execution due to an error. By default, Autolink generates a global timer *T_Global* automatically and starts it at the beginning of each test case. At the end of each test sequence, *T_Global* is cancelled. The automatic generation of timers can be enabled and disabled with the Define-Global-Timer command.

2. A delaying timer is used to delay the sending of a signal from the tester to the SUT. This may be done for several reasons; for example, the sending is delayed on purpose to specify an invalid behavior of the environment.

3. A guarding timer is used to check that the SUT sends a signal within a predefined amount of time.

Delaying and guarding timers have to be specified manually on the environment instances in test case MSCs. As an example, the MSC in Figure 254 contains a guarding timer *T_Guard* on instance *ISAP1* and a delaying timer *T_Wait* on instance *MSAP2*.

T_Guard is set prior to sending a signal to the SUT and reset after the corresponding response from the SUT. If message *ICONconf* is received in time, test execution proceeds normally. Otherwise, a timeout of *T_Guard* will be caught in the Default Dynamic Behavior description and lead to a *fail* verdict.

The setting of timer *T_Wait* is followed immediately by a timeout event, causing the tester to delay the sending of message *MDATreq*.

> **Note:**
>
> Test suite timers are part of the environment of the SUT. Correspondingly, there is no explicit relation between these timers and any timers which might be used within the system. Autolink simply translates timer set events on environment instances into TTCN `START` operations, timer reset events into TTCN `CANCEL` operations and timeout events into TTCN `TIMEOUT` events. However, timer events on any MSC instance belonging to the SUT are not translated into TTCN statements.

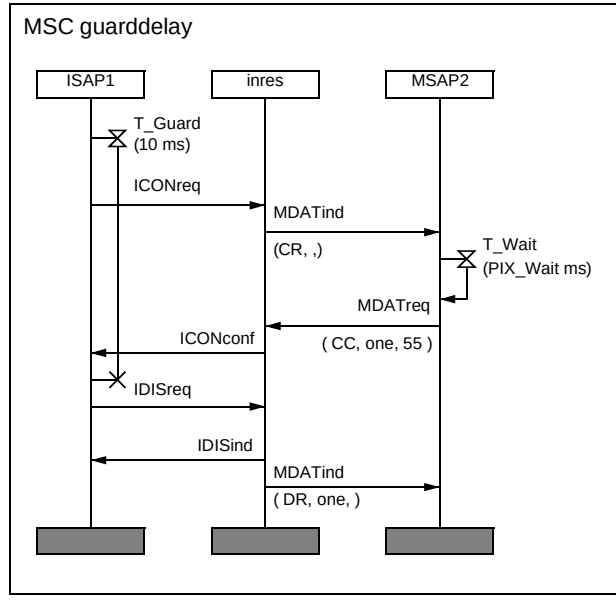


Figure 254: Test case MSC with guarding and delaying timer

Autolink creates timer declarations automatically from the information which it finds in the test case MSCs. From the MSC in Figure 254, Autolink generates a declaration for timer *T_Guard* with the default duration set to *10* and *ms* as unit. Correspondingly, the default duration for *T_Wait* is set to the test suite parameter *PIX_Wait* and its unit is set to *ms* as well. Autolink also generates a declaration for *PIX_Wait*.

This implicit declaration mechanism is convenient if the test suite consists of only one test case. If there are more test cases using timers, declaration conflicts may arise since timers are declared globally for the test suite. In order to solve this problem, Autolink provides the Define-Timer-Declaration command to explicitly declare test suite timers. Explicit timer declarations can not be modified by implicit declarations. It is therefore recommended to define all timer declarations explicate before test case computation starts. In that case, only the timer name must be provided for timer set events in the test case MSCs. The duration is optional (Autolink uses the duration value if it is not empty and different from the default duration specified in the declaration). Finally, the unit can be omitted from the test case MSCs if an explicit declaration exists.

The use of test suite timers and their declaration is explained in more detail in "Test Suite Timers" on page 1483. With respect to timers, the TTCN output generated by Autolink depends on the test architecture. This is also discussed in "Test Suite Timers" on page 1483.

Defining Multiple Test Cases by HMSC Diagrams

HMSC diagrams can be used to illustrate the relationship between various test cases. For example, even though test cases normally have different test purposes, they might share the same preamble and postamble. This commonness can be graphically expressed by the use of an HMSC diagram such as the one in Figure 255.

Note:

HMSCs are not supported by Generate-Test-Case but only by Translate-MSC-Into-Test-Case.

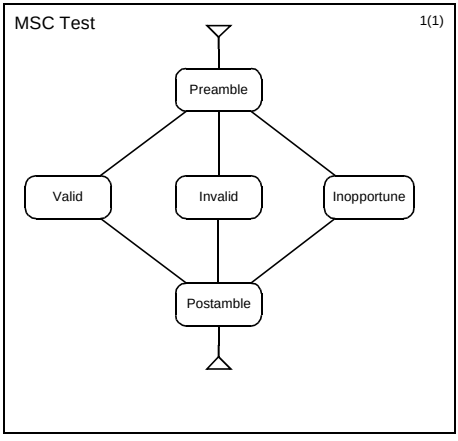


Figure 255: Three test cases described by one HMSC diagram

When the HMSC *Test* is taken as input, Autolink will create three test cases which consist of the test steps *Preamble/Valid/Postamble*, *Preamble/Invalid/Postamble* and *Preamble/Inopportune/Postamble*.

The general interpretation of HMSCs can be described by a simple rule: Autolink generates a separate test case for each possible path through an HMSC. Of course, HMSCs may have more than one node with several outgoing edges, resulting in a potentially large number of test cases.

Note:

Autolink does not translate loops directly into equivalent constructs in TTCN. Instead it handles loops by unrolling. Therefore, you should not introduce loops in HMSC diagrams.

All test cases need to have unique names. With regard to the HMSC *Test* in Figure 255, the resulting test cases will be named *Test_Valid*, *Test_Invalid* and *Test_Inopportune*. Whenever there is a branch in the HMSC, the name of the succeeding MSC reference is postfixed to the name of the top-level MSC (separated by '_').

Describing Indeterministic Behaviour by Inline and Reference Expressions

Sometimes, a system under test may not behave deterministically. For example, there may be unpredictable failures. A tester should be able to handle such situations. By the use of inline expressions and MSC reference expressions, it is possible to describe test cases where the tester reacts flexibly depending on the system behaviour.

If some of your test cases only differ slightly at some point in the test case, you may also use inline and reference expressions to describe different behaviour of the tester. In that case, Autolink generates separate test cases.

Autolink supports the following operators in MSC expressions:

- The *alternative* operator (`alt`) is suitable for the description of situations where the continuation of a test case depends on the former output of the system. If both alternatives start with a signal sent from the system to the tester (i.e. the environment), Autolink will generate two branches within a single test case.

  On the other hand, the operator may also be used to specify two alternative test sequences. If both alternatives start with a signal sent from the tester (environment) to the system, Autolink will generate two distinct test cases.

- The *optional* operator (`opt`) can be used, e.g., to accept signals which may or may not be sent by the system or to react to unexpected signals in a way that the test case can be continued normally afterwards.

- The *exception* operator (`exc`) is intended to be used for error handling. An exception expression may contain signals which prevent the test system from continuing the regular test execution. In MSC *ConnectionRequest* (Figure 256 on page 1442), the reception of signal *IDISind* immediately stops the test case. Optionally, an exception includes a sequence of signals which bring the system under test back into a stable testing state. An exception always results in an “INCONC” verdict.
- The *loop* operator (`loop`) can be used to describe the iterative execution of a (portion of a) test case. As in HMSCs, Autolink does not translate loop expressions directly into equivalent constructs in TTCN. Instead, it handles loops by unrolling. If no upper loop boundary is given, the loop is evaluated up to three times.
- Finally, the *sequence operator* (`seq`) can be used within reference expressions in order to state that one test step follows another.

Note:

Autolink does not support the *parallel* (`par`) and *substitution* (`subst`) operator within inline and reference expressions.

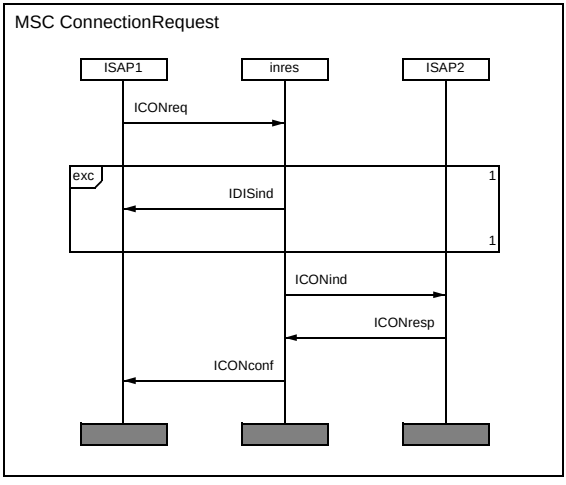


Figure 256: Test case with exception handling

Of course, the usage of the different operators is not restricted to the applications described above. On the other hand, not all MSCs containing inline or reference expressions may describe sensible test cases.

Synchronization among MSC expressions

As mentioned before, Autolink synchronizes at the beginning and the end of MSC references. This modification of the semantics given in the MSC standard is motivated by the ability to consider separate MSCs as different test steps. If we cannot guarantee that all events in one MSC are evaluated before the next MSC is entered, we cannot draw a line between two subsequent test steps.

MSC reference expressions are a generalization of plain MSC references. Autolink generates distinct TTCN test steps for each of the MSCs involved in the expression. For that reason, it makes sense to synchronize at MSC references, too.

When it comes to inline expressions it is not really necessary to synchronize at their beginning and their end. However, for consistency Autolink synchronize at inline expressions, as well.

There are situations where synchronization among inline expressions is preferable, whereas in other cases the TTCN output and Autolink's error messages may be confusing.

In Figure 257, two examples are given. For MSC *Sync1*, Autolink will generate a test case where either *ReceiveB* or *ReceiveC* is anticipated before the test execution proceeds with *SendD*. If Autolink did not synchronize at the end of the alternative expression, *SendD* would be sent before *ReceiveB* (please note that Autolink prioritizes send events). Moreover, since both alternatives are combined in a single behaviour tree, *SendD* would erroneously become an alternative to *ReceiveC*!

Unfortunately, there are also cases where synchronization results in unexpected TTCN test cases. For the second MSC in Figure 257, Autolink will try to generate the following event tree:

```
Env1 ! SendA
  Env2 ? OptReceiveB
    Env1 ! SendC
      Env2 ? ReceiveD
  Env1 ! SendC
    Env2 ? ReceiveD
```

Of course, Autolink detects that there is a conflict among the two alternatives *OptReceiveB* and *SendC* and will print a warning.

Whenever Autolink issues a warning, you should carefully inspect your MSC test case definition. For example, when MSC *Sync2* is interpreted as a test case, it is unclear how long a tester shall wait until it outputs *SendC*. On the other hand, if *OptReceiveB* is not logically caused by *SendA* or may received at any time, a possible solution would be to move *SendC* above the optional expression. In this case, no conflict arises.

Synchronizing Test Events with Conditions

The messages in MSCs are only partially ordered. If a test generation tool would generate all possible test sequences, then send events could appear as alternatives to receive events in TTCN test cases, making them indeterministic. To solve this problem, three solutions are possible:

1. Events received by the tester are prioritized over events sent to the SUT.
2. Events sent to the SUT are prioritized over events received by the tester.
3. All messages in the MSC are evaluated from top to bottom. In that case, only one sequence of test events is generated.

The first alternative may lead to deadlocks and therefore it is not supported by Autolink. Alternative three may be selected with the Define-Autolink-Generation-Mode command.

By default, Autolink uses the second alternative and prioritizes events which are sent from the test system to the SUT over events which are received by the tester from the SUT.

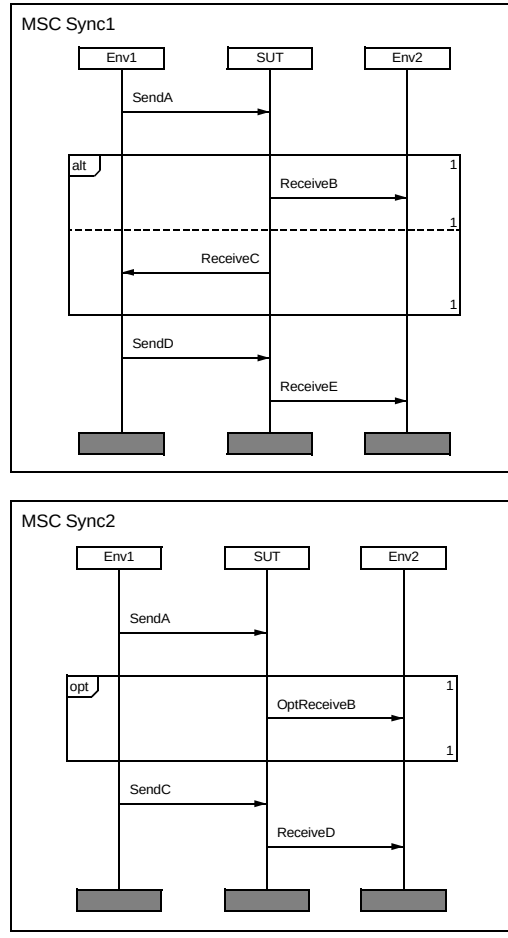


Figure 257: Synchronization among inline expression

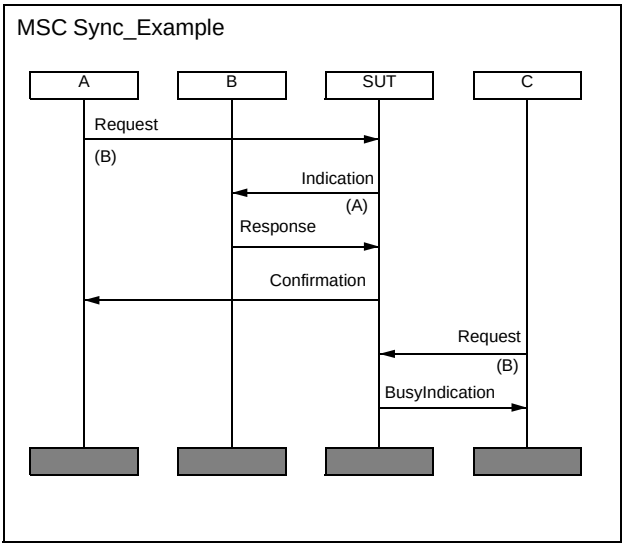


Figure 258: Synchronization example 1

However, there are situations where this “send immediately” strategy leads to incorrect test cases. Consider the following small example: A calls B, B accepts the call, then C tries to call B and gets a busy signal. A corresponding MSC test purpose description would look similar to the one in Figure 258. What test sequence will Autolink generate? First, the tester will send `Request(B)` to the SUT through PCO A. Next, `Request(B)` will be sent to the SUT through PCO C, since this is the next send event on any of the tester instances.

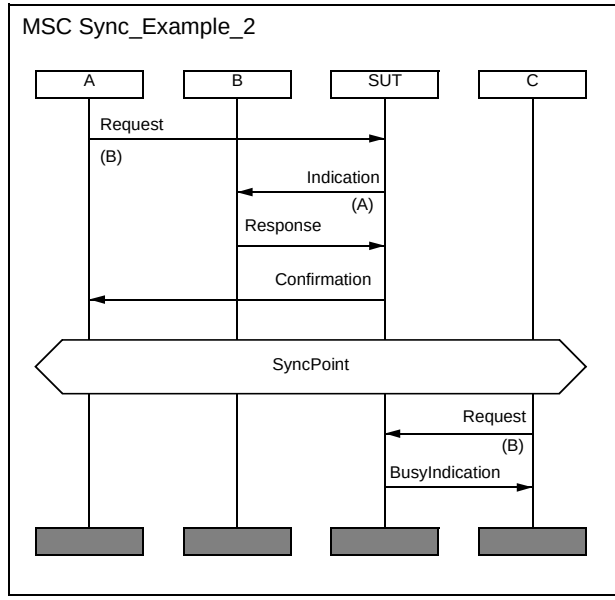


Figure 259: Synchronization example 2

Most likely, this is not what you have anticipated intuitively; the tester should send the second `Request(B)` only after it has received `Confirmation` through PCO A. This is where explicit synchronization with conditions comes in. Figure 259 contains the same MSC as Figure 258, with a global condition added between the reception of `Confirmation` at PCO A and the sending of `Request(B)` through PCO C. If the Autolink synchronization option is turned on (with Define-Condition-Check), then events below the condition can only be executed if all events above the condition have been executed as well. The condition effectively becomes a synchronization point. In the example, the sending of `Request(B)` through C will be delayed until `Confirmation` has been received at PCO A. The name in the condition is required by the ITU Recommendation Z.120, however it is only considered as a comment and does not have any semantics.

Note:

The condition check option also applies to the MSC verification function of the Explorer.

Listing and Clearing Test Cases and Test Steps

- Use List-MSC-Test-Cases-And-Test-Steps for listing all MSC test cases and test steps that are defined in the current test cases and test steps directories.
- Use Clear-MSC-Test-Case to delete an MSC test case.
- Use Clear-MSC-Test-Step to delete an MSC test step.

Note:

Autolink distinguishes between the MSC test case name (which is the name of the system level MSC) and the name of the file on disk that contains the MSC test case. Usually, these names are identical except for the file extension.

When accessing the test cases and the test steps directory, Autolink always refers to the file names, meaning that you also have to specify the file extension (either `.mpr` or `.msc`)

Defining MSC Test Cases Automatically - Coverage Based Test Generation

One common method to generate a test suite is to select a number of test cases which, taken together, obtain a high coverage of the SDL specification. Ideally, this coverage should be 100%. In that case, every SDL symbol in the specification is executed at least once during test generation.

Autolink provides a special state space exploration technique, called *Tree Walk*, which is optimized for finding paths through the state space of the SDL specification that result in a high SDL symbol coverage. Tree Walk combines the advantages of both the depth-first and breadth-first search strategy - it is able to visit states located deep in the reachability graph and to find a short path to a particular state at the same time.

- To use Tree Walk, apply the Tree-Walk command.

The execution of Tree Walk is controlled by two options:

1. The maximum computation time (specified in minutes)
2. The targeted coverage (specified in percent)

When Tree Walk is used, it generates a report for each sequence of transitions in the state space that increases the number of visited SDL symbols. These reports can be converted into system level MSCs with the command Save-Reports-as-MSC-Test-Cases.

Defining an Autolink Configuration

Autolink offers a special language for defining rules for the naming and parameterization of constraints, the introduction of test suite parameters and constants, and the distribution of test cases and test steps into test groups.

- To specify Autolink configuration rules, use the command Define-Autolink-Configuration.

While it is possible to define an Autolink configuration on the fly at the Explorer command prompt, it is better to write a command file which includes the configuration.

- To load this file, use the Include-File command.

  The command corresponds to the menu choice *Configuration: Load* in the *Autolink2* menu.

- To remove a loaded configuration, use the Clear-Autolink-Configuration command.
- To display the current configuration, use the Print-Autolink-Configuration command.
- To save a configuration, use the Save-Autolink-Configuration command.

See also:

- “Translation Rules” on page 1459 for an introduction to translation rules.
- “Test Suite Structure Rules” on page 1465 for the methodology for test suite structure rules.

- "Syntax and Semantics of the Autolink Configuration" on page 1471 for a detailed description of the configuration language.

Computing Test Cases

Once you have defined a set of MSC test cases, you can compute internal test case representations for each MSC test case.

- To compute MSC test cases, use the Generate-Test-Case command.

  A *bit-state exploration* will be started which aims at finding all possible sequences of observable events that conform to the MSC. In addition, Autolink searches for *inconclusive events*. These are events that represent deviations from the behavior specified in the MSC test case, but which are valid alternatives according to the SDL specification.

Either a single test case or a set of test cases in the current test cases directory may be generated at the same time, depending on the parameter of Generate-Test-Case. If an already generated test case is regenerated, its former internal representation and its corresponding constraint definitions are replaced.

> **Note:**
>
> If the SDL specification is not detailed enough in the sense that it does not model the signal flow in a given MSC, the generation of the test case fails. MSCs which cannot be verified can still be converted into test cases with the Translate-MSC-Into-Test-Case command, which is described in "Translating MSCs into Test Cases" on page 1454.
>
> Inline expressions, MSC reference expressions and HMSC diagrams are not supported by Generate-Test-Case. (However, you can use these structural concepts for Translate-MSC-Into-Test-Case.) If you want to generate test cases based on HMSCs, you have to transform your HMSC diagrams into basic MSCs first. You can do this by verifying the HMSCs and saving all *MSC Verification* reports as MSC test cases (command Save-Reports-as-MSC-Test-Cases).

With complex systems, test generation may take a while. To avoid time-consuming test generation failures, you should verify all MSCs first.

- To verify all MSC, use the Verify-MSC command.

In most cases, MSC verification takes only a fraction of the time needed for test generation.

Listing and Clearing Generated Test Cases

- You can list all generated test cases with the List-Generated-Test-Cases command.
- You can clear all generated test cases with the Clear-Generated-Test-Case command.

Displaying and Saving the Internal Representation

- You can display the internal representation of a generated test case – without having to save the test suite and start the TTCN Suite – with the Print-Generated-Test-Case command.
- You can save an internal test case representation with the Save-Generated-Test-Case command.
- You can load an internal test case representation back into memory with the Load-Generated-Test-Cases command.

This makes it possible to distribute the generation of a set of test cases to several computers. When all test cases have been generated and saved in individual files, they can be reloaded on a single machine and saved as a complete test suite.

State Space Exploration Parameters

There are several parameters which influence the state space exploration:

- You can set the maximum search depth with the Define-Autolink-Depth command (default value is 1000).

  If the search depth is set too low, this may result in incomplete pass paths. Since the maximum search depth normally has no impact on the performance of the state space search, it is recommended that you always use a large value (> 1000).

- You can resize the hash table with the Define-Autolink-Hash-Table-Size command (default size is 1,000,000 bytes).

  Increasing the hash table size may be necessary if the SDL system is rather complex.

- Normally, Autolink changes the regular Explorer state space options, because the default values of the Explorer are not sufficient for generating correct test cases.

  For example, for the Inres protocol specification, the following commands are issued:

  ```
  Define-Scheduling All
  Define-Transition Symbol-Sequence
  Define-Symbol-Time Zero
  Define-Priorities 2 1 3 2 2
  Define-Channel-Queue ISAP1 On
  Define-Channel-Queue MSAP2 On
  Define-Max-Input-Port-Length 10
  ```

  Other SDL systems require different Define-Channel-Queue commands.

  Autolink automatically resets the original options when a test case generation is finished. Therefore, previously defined reports are still valid (unless they were generated by verifying an MSC).

  If you do not want to use the default settings for any reason, use the Define-Autolink-State-Space-Options `Off` command. This disables the automatic setting of Autolink's default options.

Test Case Generation Messages

After the generation of a test case, some warnings or error messages may be displayed:

- `Error: Autolink did not find any complete pass paths. No test case is generated.`

  This message appears if no final TTCN *pass* verdict has been assigned to any event in the internal test case representation.

  The most common reason for test generation failure is that the MSC does not describe a valid trace of the SDL system. Hence, before running Autolink, you should check that the MSC test case can indeed be verified using command Verify-MSC.

  The error message above may also appear if the maximum search depth is set too low. Check the state space exploration statistics which are displayed by Autolink at the end of the test case generation. If the number of truncated paths is greater than zero, you should increase the maximum search depth. The required value for

the search depth depends both on the length of the test case and on the selected state space options of the Explorer.

Autolink may also fail if you do not specify the signal parameters in the MSC test cases, but instead use global test values. Make sure that at least all signals sent *into* the system are fully specified in the MSC.

- `Warning: Incomplete pass path found.`
  `The first event X on the incomplete branch`
  `gets an 'INCONC' verdict.`

  This message is displayed if a path in the state space is pruned before a final TTCN *pass* verdict has been assigned to an observable event. (Note that in this context a *pass path* refers to the internal test case representation whereas otherwise, a path refers to the state space of the SDL system.) If this happens, the first event *X* of the incomplete subtree is reassigned as a TTCN *inconc* verdict, and the rest of the subtree is discarded.

  The reasons for a pass path being pruned are numerous. The system level MSC may not be verifiable, for example if the state space is too restricted, or there may be a path in the state space of the SDL system that only partially verifies the MSC. A common problem is the maximum search depth being too low (see above).

- `Warning: Alternative events found for send event.`
  `Inconclusive events are deleted.`

  For test generation, it is assumed that signals which can be sent to the system are sent instantaneously. Therefore, alternatives to send events are not desired.

  With the default options of Autolink, this message should not appear. If you use your own Autolink options, check whether *Input from ENV* has the highest priority. To correct the problem, you should use the Define-Priorities command with parameters `X1 1 X2 X3 X4`, where $X_i > 1$.

- `Warning: No translation rule can be applied to the following signal: <Signal Name>`

  If you define an Autolink configuration containing translation rules, Autolink assumes that you want to map *all* SDL signals onto constraints with the help of user-defined rules. Hence, you will be

warned if there is a signal in a test case to which no translation rule can be applied.

- `Warning: There are events following an event with final 'PASS' verdict.`

  When the MSC test case has been simulated completely, the SDL system was not yet in a stable state. Instead, it sent out one or more additional signals which were not specified in the MSC test case. You should carefully review your test specification, since during a test campaign you may not be able to successfully execute another test case after the execution of the faulty test case.

Translating MSCs into Test Cases

If an SDL system is not fully specified, some or all MSCs may not be converted into test cases if you use the Generate-Test-Case command. Instead, you can use the Translate-MSC-Into-Test-Case command to translate MSCs directly into the same internal test case format which is used for test cases generated by state space exploration. Therefore, you can use all commands related to listing, displaying, removing, saving and loading of test cases (introduced in "Computing Test Cases" on page 1450) with directly translated test cases as well.

Furthermore, all rules for constraint naming (see "Translation Rules" on page 1459) and test grouping (see "Test Suite Structure Rules" on page 1465) apply to translated test cases, too.

> **Note:**
>
> Since the translation of MSCs into test cases does not perform a state space exploration, there is no guarantee that the MSCs and hence the test cases describe valid traces of the specification or the implementation, respectively. Instead, the validity of the test cases has to be assured by the developer. Furthermore, no *inconclusive* events can be computed during MSC translation. Therefore, the resulting test cases return a *fail* verdict for any deviation from the behavior described in the MSC.

Either a set of test cases in the current test cases directory or just a single one may be translated at the same time, depending on the parameter of Translate-MSC-Into-Test-Case. If an already translated test case is retranslated, its former internal representation and its constraint definitions are replaced.

Two different algorithms are available for MSC translation: With the Define-Autolink-Generation-Mode command, you can choose the semantics used to interpret MSCs during translation. By default, Autolink uses the standard semantics of MSC. If the generation mode is set to “total ordering”, then the sequence of input and output events in the MSC is determined straight from top to bottom. If there are two events on different environment instances, Autolink evaluates the event which is closest to the top of the MSC first.

> **Note:**
>
> For test generation with state space exploration (using the Generate-Test-Case command), total ordering is not supported.

MSC into TTCN Translation Messages

After the generation of a test case, some warnings or error messages may be displayed:

- `Error: No test case could be generated. Please check whether the MSC contains separate instance axes for each channel to the environment.`

  The Translate-MSC-Into-Test-Case command does not support the translation of MSCs with only one instance axis for the environment. Use MSCs with a separate instance axis for each channels to the environment (i.e. for each PCO).

- `Warning: No translation rule can be applied to the following signal: <Signal Name>`

  If you define an Autolink configuration containing translation rules, Autolink assumes that you want to map *all* SDL signals onto constraints with the help of user-defined rules. Hence, you will be warned if there is a signal in a test case to which no translation rule can be applied.

Modifying Constraints

It is highly recommended that you specify constraints naming and parameterization rules in an Autolink configuration file. Otherwise, a generic name of the form <Test case name>_<three digit number> is assigned to each constraint during test case generation. However in the latter case, you will probably find it useful to modify the constraints generated by Autolink.

- Use the Rename-Constraint command to change the name of a constraint.

  Besides renaming a constraint, it is also possible to merge two constraints. Do this by giving a constraint the same name as another one. Then you will have to select which of the two signal definitions should be kept (unless they are identical). There is one restriction: Constraints with formal parameters cannot be overwritten by other constraints.

  See also "Translation Rules" on page 1459.

- Use the Merge-Constraints command to merge two constraints by potentially introducing formal parameters.

  If the original constraints are used in test cases, their constraint references will be updated. This means that the concrete signal parameters (which were replaced by formal parameters) will be moved to the constraint references.

Note:

If you rename a constraint or merge two constraints, the internal test case descriptions are updated, too. The links between the events in the test cases and their corresponding constraints remain consistent.

- Use the Define-Constraint command to add new constraints to the current list of constraints.

  If a constraint with the same name but a different signal definition already exists, you will have to choose what to do – rename the new constraint, overwrite the old constraint or remove the new definition.

- Use the Parameterize-Constraint command to replace concrete signal parameter values in a constraint by formal (symbolic) parameters. If the parameterized constraint is used in a test case, the parameter value is not lost, but maintained in the constraint references of the referring test cases instead.

- Use the Clear-Constraint command to delete a constraint.

- Use the List-Constraints command to list all currently defined constraints.

- Use the Save-Constraint command to save one or all constraints.

- Use the Load-Constraints command to reload saved constraints.

Generating a TTCN Test Suite

- Use the Save-Test-Suite command to save a test suite in a TTCN-MP file.

  The internal representations of the test cases will be kept in memory in order to allow you to save test suites in different formats.

- There are two fundamentally different formats to save test suites: "Traditional" TTCN and concurrent TTCN. To switch between these formats, use the ***Define-Concurrent-TTCN*** command. For a detailed description of the concurrent TTCN format, see "Concurrent TTCN" on page 1478.

- By default, constraints are stored as ASN.1 ASP constraints, but before you generate a test suite, you may change an option to have them stored as ASN.1 PDU constraints instead. To do this, use the Define-TTCN-Signal-Mapping command. You are also allowed to select the correct type of constraint for each signal individually by adding rules to an Autolink configuration (see "Defining ASP and PDU Types" on page 1469).

- There are three possible output formats for test steps. Use the Define-TTCN-Test-Steps-Format command to select an output format:
  - One possibility is to store the test steps of a single test case as *local trees*. If a test step is used several times, only one behavior description is generated.
  - Test steps can also be stored globally in the *test step library*. If a test step is used several times in different test cases, only one behavior description is generated.
  - A third alternative is to generate "flat" test cases by including the events of the test steps directly in the test case dynamic behavior descriptions. In this case, no information about test steps is put into the TTCN test suite.

Note:

A test step that is used in several places may lead to trees with different inconclusive events or different verdicts. In this case, they will be given new, unique names.

Preliminary Pass Verdicts

Test cases that are structured into preamble, test body and postamble will automatically be assigned preliminary pass verdicts at the end of the test body. However, test cases can contain an arbitrary number of MSC references (and hence test steps). Therefore, preliminary pass verdicts will be assigned to all events that are directly followed by the last top-level MSC reference in the test case. The preliminary pass verdicts will only be assigned if no event follows the last MSC reference. The event to which a preliminary pass verdict is assigned may appear within the test body as well as within a test step.

Test Suite Generation Messages

During the saving of a test suite, some warnings or error messages may be displayed:

- `Warning: Test step <TS> resulted in different trees. The trees are renamed to '<TS>_1', '<TS>_2', etc. in the test suite.`

  If an MSC test step is reused in several test cases, the resulting TTCN test steps may be different. Typically, this warning appears if a test step is used as a preamble in one test case and then again as a complete test case by itself. In the latter case, a final pass verdict is assigned to the test case, while in the former one it is not.

- `Warning: No test suite structure rule defined for test case/step '<TestCaseName/TestStepName>'.`

  If you define an Autolink configuration containing test suite structure rules, Autolink assumes that you want to place *all* test cases/steps in test groups defined by the test suite structure rules. Hence, you will be warned if there exists a test case/step to which no rule can be applied.

- `Warning: Test suite parameter/constant '<Name>' is not unique.`
  `It is renamed to '<Name>_1', '<Name>_2', etc. in the test suite.`

  By using translation rules (see "Translation Rules" on page 1459) you can introduce test suite parameters and constants. These parameters and constants are checked for consistency in a similar way as test suites. With the warning above, Autolink informs you that it had to rename test suite parameters/constants in order to resolve naming conflicts.

Translation Rules

In "Modifying Constraints" on page 1455, you have learned how to change constraints. However, assigning sensible names to automatically generated constraints is a tedious task. Especially if you have to refine the SDL specification and then to repeat the test generation process, there is a lot of manual work. Moreover, the number of generated constraints may become very large if you do not use constraint parameterization.

In order to address these problems and some additional issues, you can specify so-called *translation rules*. These rules control the look of a test suite with regard to the following items:

1. Naming of constraints
2. Parameterization of constraints
3. Replacement of signal parameter values by wildcards in a constraint declaration table
4. Introduction and naming of test suite parameters and test suite constants

Translation rules build one integral part of an Autolink configuration (see also "Test Suite Structure Rules" on page 1465). Before you start the test generation, you can develop an Autolink configuration file that contains a Define-Autolink-Configuration command. The set of translation rules which tell Autolink how to construct constraints and treat parameters for particular signals, are provided as a kind of long parameter to this command.

For some examples, see "Examples of Translation Rules" on page 1459. More information can be found in "Defining an Autolink Configuration" on page 1449 and "Syntax and Semantics of the Autolink Configuration" on page 1471.

Examples of Translation Rules

A typical translation rule may look like this:

Example 274 ––

```
TRANSLATE MDATind
  CONSTRAINT NAME "C_" + $0
             PARS $1="Type"
```

```
END
-------------------------------------------------------------
```

Example 274 explains how signal `MDATind` is translated into an suitable TTCN constraint. The rule above states that the name of a constraint for signal `MDATind` consists of the concatenation of text `"C_"` and the "nullth" parameter – which is the name of the signal itself. Therefore, signal `MDATind` is translated into a constraint called `C_MDATind`.

Additionally, the first parameter of the signal (referred to by `$1`) becomes a parameter of the constraint. The name of the formal parameter is `Type`. It is printed both in the *Constraint Name* line and the *Constraint Value* section of the constraint declaration table. The actual parameter of the constraint is printed in the dynamic behavior table of each test case that uses this constraint.

A constraint declaration table for signal `MDATind` is shown in Figure 260.

ASN.1 ASP Constraint Declaration	
Constraint Name :	C_MDATind(Type : IPDUType)
ASP Type :	MDATind
Derivation Path :	
Comments :	
Constraint Value	
{ iPDUType1 Type, sequencenumber2 zero, iSDUType3 0 }	
Detailed Comments :	

Figure 260: A constraint declaration with a formal parameter

It is also possible to define a single translation rule for more than one signal. This is especially useful if similar signals exist which can be treated in the same way.

Example 275 ----------

```
TRANSLATE MDATind | MDATreq
  CONSTRAINT NAME "C_" + $0
             PARS $1="Type"
END
```

If either `MDATind` or `MDATreq` is identical to the signal for which a constraint is to be created during test generation, the rule in Example 275 is applied. The value of `$0` depends on the name of the actual signal investigated at run-time. Since the first signal parameter is always to be replaced by the formal parameter `Type`, the rule is only valid if each of the alternative signals, i.e. `MDATind`, and `MDATreq`, has at least one parameter. When parsing an Autolink configuration, all translation rules are checked automatically for validity.

Constraint names may not only be based on texts and signal names. They can also depend on signal parameters. In a translation rule, a signal parameter is referred to by its number, prefixed with a dollar character (`$`). (Note that Autolink only supports parameters on the top level – it is not possible to refer to a component of a nested parameter.)

In some cases, it is not desirable to use the value of a signal parameter directly as part of a constraint name. For example, a protocol engineer might encode complicated signal information with abbreviations or numbers. But for the TTCN output, parameter values should be mapped onto more meaningful expressions.

Therefore, you may define functions which take an arbitrary number of parameters and map them onto text. In Example 276, the value of the first parameter of signal `MDATind` is passed to function `PDUType`. Depending on the concrete parameter value, which occurs during test case generation, the function returns a text. This text forms the second part of the constraint name.

Example 276 ----------

```
TRANSLATE MDATind
  CONSTRAINT NAME "Medium_" + PDUType($1)
END

FUNCTION PDUType
    $1 == "CR"  : "Ind_Connection_Request"
  | $1 == "AK"  : "Ind_Acknowledge"
  | $1 == "DR"  : "Ind_Disconnection_Request"
  | TRUE        : "Indication"
END
```

Note:

In a translation rule, `$i` refers to the i-th parameter of the signal for which a constraint is created. However in a function, `$i` denotes the i-th parameter which was passed to the function.

You may define complex rules whose evaluation is guarded by conditions. This is illustrated in Example 277.

Example 277 ----------

```
TRANSLATE "MDATind"
  IF $1 == "CR" THEN
    CONSTRAINT NAME "Medium_Connection_Request"
  END
  IF $1 == "AK" AND $2 == "zero" THEN
    CONSTRAINT NAME "Medium_Acknowledge_Zero"
  END
  CONSTRAINT NAME "Medium_Indication"
             PARS $1="Type"
END
```

Conditional translations can be defined by IF-statements. Only if the condition(s) following the `IF` keyword is/are satisfied, the constraint is built according to the subsequent specification. A translation rule can contain several `IF`-clauses. The first clause which condition is satisfied (or which does not have an `IF` statement at all) is chosen for translation.

In the example above, signal `MDATind` is translated into a constraint called `Medium_Connection_Request` if the first signal parameter equals `CR`, and to a constraint called `Medium_Acknowledge` if the first two signal parameters equal `AK` and `zero` respectively. If neither condition is satisfied, the unconditioned section is evaluated. In this case, a constraint with name `Medium_Indication` and formal parameter `Type`

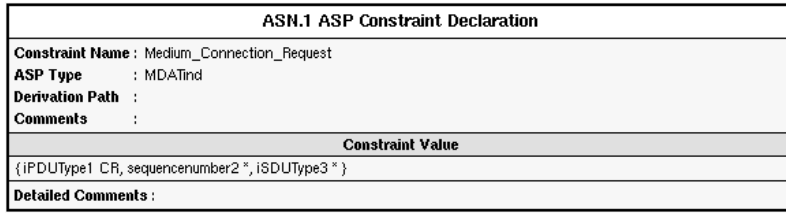
is created. Note that the parameter definition is not taken into account if any of the former IF-conditions is satisfied!

Sometimes, it is useful to indicate that a specific signal parameter is irrelevant. For example, assume that if the first parameter of signal MDATind is CR, the values of the second and third parameter can be ignored. Hence, you can replace them by wildcards in a constraint table. In Example 278, a MATCH statement is added that tells Autolink to replace the values of the signal parameters 2 and 3 by asterisks. The resulting constraint table is displayed in Figure 261.

Example 278 ----------

```
TRANSLATE "MDATind"
  IF $1 == "CR" THEN
    CONSTRAINT NAME "Medium_Connection_Request"
               MATCH $2="*", $3="*"
  END
  CONSTRAINT NAME "Medium_Indication"
             PARS $1="Type"
END
```

> **Note:**
>
> The application of TTCN matching mechanisms is only valid for receive events. Hence, you are not allowed to apply the MATCH statement to signals that become send events in TTCN.



ASN.1 ASP Constraint Declaration	
Constraint Name :	Medium_Connection_Request
ASP Type :	MDATInd
Derivation Path :	
Comments :	
Constraint Value	
{iPDUType1 CR, sequencenumber2 *, iSDUType3 * }	
Detailed Comments :	

Figure 261: A constraint table with TTCN matching expressions

Translation rules also allow to introduce test suite parameters and constants. Test suite constants are useful if a concrete parameter value does not give any clues about its meaning and hence should be replaced globally by a more meaningful name. Test suite parameters should be introduced if signal parameter values are implementation dependent. By defining a test suite constant/parameter, a concrete signal parameter

value in a constraint table is replaced by a symbolic constant. The assignment of concrete values to symbolic test suite constants/parameters is made in additional TTCN tables which are created automatically by Autolink.

Example 279 illustrates the use of test suite parameters and constants. If the condition is satisfied, the second signal parameter is replaced globally by `SeqNo` in the TTCN test suite. The third signal parameter is replaced by a test suite parameter called `DataValue`. This parameter refers to PICS/PIXIT proforma entry `PICS_Data`.

If signal MDATreq has not been used for data transfer, the value of the first signal parameter is replaced by a test suite constant which name is based on the concrete signal parameter value. A constraint table and an according constant table for this case is shown in Figure 262 and Figure 263.

Example 279 ----------

```
TRANSLATE "MDATreq"
  IF $1 == "DT" THEN
    CONSTRAINT NAME   "Medium_Req_Data_Transfer"
    TESTSUITE  CONSTS $2="SeqNo"
               PARS   $3="DataValue" / "PICS_Data"
  END
  CONSTRAINT NAME   "Medium_Req_" + PDUType($1)
             MATCH  $3="*"
  TESTSUITE  CONSTS $1=PDUType($1)
END

FUNCTION PDUType
  $1 == "CC" : "ConConf"
| $1 == "AK" : "Acknowledge"
| $1 == "DR" : "DisconRequest"
| $1 == "DT" : "DataTransfer"
| $1 == "CR" : "ConRequest"
END
```

ASN.1 ASP Constraint Declaration	
Constraint Name :	Medium_Req_ConConf
ASP Type :	MDATreq
Derivation Path :	
Comments :	
Constraint Value	
{ iPDUType1 ConConf, sequencenumber2 one, iSDUType3 * }	
Detailed Comments :	

Figure 262: A constraint table with a test suite constant

Test Suite Constant Declarations			
Constant Name	**Type**	**Value**	**Comments**
Acknowledge	IPDUType	AK	
ConConf	IPDUType	CC	
Detailed Comments :			

Figure 263: A TTCN table for test suite constant declarations

An Autolink configuration typically consists of a large number of translation rules which are evaluated from top to bottom. If a constraint cannot be constructed based on the given rules, a generic name will be assigned to the constraint, in the same way as when no translation rules are defined.

Test Suite Structure Rules

In TTCN, test cases can be combined in test groups. Each test group aims at testing the system under test for one particular aspect. Test groups again can be part of other higher level test groups, resulting in a hierarchy of test groups.

Test steps can be put into test groups as well. In the following, test cases and test steps will not be distinguished, as test structure rules apply to both.

When you start designing a test suite, you should have a clear notion of what the structure of the test suite will be. In fact, for successful test suite development, it is important to first determine what should be tested and how the tests can be classified, before individual test cases are specified.

If you use Autolink for test generation, the test cases are described by MSCs. Ideally, the names of the MSCs should give information about

the structure of the resulting test suite. Because of this, you may specify rules for the automatic placing of test cases in different test groups, depending on the names of the corresponding MSCs. These *test suite structure rules* prevent you from repeating a lot of manual work if you regenerate the test suite due to a modification of the underlying SDL specification. Moreover, test suite structure rules (TSS rules) also save you a lot of work if you create a test suite only once, since a single rule can be applied to several test cases. As will be shown in the example below, one rule may be enough to describe the structure of a complete test suite.

Test suite structure rules are part of an Autolink configuration. Before the test generation starts, you can write an Autolink configuration file which contains a Define-Autolink-Configuration command. The TSS rules are provided as a kind of long parameter to this command.

For details on the Autolink configuration commands see "Defining an Autolink Configuration" on page 1449. A precise description of the Autolink configuration language is given in "Syntax and Semantics of the Autolink Configuration" on page 1471.

Examples of Test Suite Structure Rules

In the following, it is assumed that you want to create a test suite in which test cases can be classified according to three different criteria. On the top level, tests can be distinguished by whether they are related to mandatory or optional capabilities. On the next level, tests may focus on particular protocol phases, for example connection establishment, data transfer and disconnection. Finally, valid, invalid or inopportune behavior may be displayed. A resulting test suite should have the following structure:

```
Mandatory
  Connection
    Valid
    Invalid
    Inopportune
  DataTransfer
    Valid
    ...
  Disconnection
    ...
Optional
    ...
```

It is further assumed that having this structure in mind, you have created MSC test cases with the following names:

```
V_Con_Man_01
V_Dis_Man_01
IV_Data_Man_01
IO_Data_Opt_01
IO_Data_Opt_02
```

MSC test cases that belong to the same test group are numbered sequentially.

Now, a simple TSS rule for the scenario above may look like this:

Example 280 ----------

```
PLACE V_Con_Man_01
  IN    "Mandatory" / "Connection" / "Valid"
END
```

Example 280 states that test case `V_Con_Man_01` is intended to be placed in the test group `Valid`. Since this test group is placed in another test group (`Connection`), you have to specify the complete path, composed of all groups in hierarchical order. The names of the test groups are separated by a slash (‘`/`’) in analogy to the notation of test group references in the TTCN standard.

If you want to place several test cases in the same test group, you can use the alternative operator (‘`|`’) in the header of a TSS rule:

Example 281 ----------

```
PLACE IO_Data_Opt_01 | IO_Data_Opt_02
  IN    "Optional" / "DataTransfer" / "Inopportune"
END
```

Example 281 places both `IO_Data_Opt_01` and `IO_Data_Opt_02` in test group `Optional/DataTransfer/Inopportune`.

Rules like the one shown in Example 280 and Example 281 can be applied to MSC test cases with arbitrary names. In the best case, you have to write one TSS rule for each test group.

However, there is a direct relation between the MSC names and the test groups. For example, the two characters `IO` at the beginning of an MSC name indicate that the corresponding test case has to be placed in a test group called *Inopportune*. Using this information, the number of TSS rules can be further reduced as explained below.

You are allowed to use patterns in the header of a test suite structure rule. The following characters have a special meaning when used in the header:

- '*' matches zero or more arbitrary characters.
- '?' matches exactly one arbitrary character.
- "[...]" matches any single character in the enclosed lists. In order to represent characters ranges, you can type two characters separated by a dash ('-'). For example, "[a-z]" denotes an arbitrary lowercase letter. If the first character is a '!', any character not enclosed is matched.

Note:

Patterns can also be used in a similar way in the header of translation rules. This is useful if signals with similar names are to be treated equally.

Now consider the following complex rule and its auxiliary functions:

Example 282 ----------

```
PLACE "*" + "_" + "*" + "_" + "???" + "_" + "*"
  IN    OptMan( @5 ) / Phase( @3 ) / Behavior( @1 )
END

FUNCTION OptMan
  $1 == "Opt" : "Optional"
| $1 == "Man" : "Manual"
END

FUNCTION Phase
  $1 == "Con"  : "Connection"
| $1 == "Data" : "DataTransfer"
| $1 == "Dis"  : "Disconnection"
END

FUNCTION Behavior
  $1 == "V"  : "Valid"
| $1 == "IV" : "Invalid"
| $1 == "IO" : "Inopportune"
END
```

With the rule in Example 282, *all* test cases can be placed in their appropriate test groups.

When a test suite structure rule is evaluated, it is first checked whether one of the terms following the keyword `PLACE` (which are separated by '|') equals the name of the investigated test case. In the rule above, there is only one term consisting of 7 parts, called atoms. These atoms are concatenated by the '+' operator.

While Autolink simply has to compare strings in Example 280 and Example 281, it has to find out whether a concrete test case name matches the pattern in Example 282. If the test case name matches the pattern, the atoms in the header of the TSS rule are instantiated.

If, for example, the rule is applied to test case `IO_Data_Opt_01` at run-time, the first atom (originally '*') is set to "IO". The value of the third atom becomes "Data", the value of the fifth atom becomes "Opt" and the value of the seventh atom becomes "01". The second, fourth and sixth atom remain unchanged as they do not contain any special characters.

In order to refer to the value of an atom in the rule header, you can use the "at" operator ('@'). For example, "@5" refers to the value of the fifth atom.

Additionally, you may define functions which map parameters onto texts. In Example 282, "@5" is passed to function `OptMan`. Depending on the concrete parameter value which is passed at run-time, the function returns either the text "Optional" or "Manual" (or an error message if the first function parameter is neither "Opt" nor "Man").

An Autolink configuration typically consists of a number of TSS rules which are evaluated from top to bottom. If a test case or a test step cannot be placed in a test group based on the given rules, Autolink places it on top-level and prints an error message. In this case, you can modify your rules, reload them and apply the Save-Test-Suite command again.

Defining ASP and PDU Types

When Autolink produces a TTCN test suite it creates several tables in the declarations part. These tables store information about sorts, ASN.1

data types and signal definitions used in the SDL system. By default, Autolink applies the following rules:

1. SDL sort definitions are mapped onto *ASN.1 type definitions*.
2. ASN.1 data types defined externally in an ASN.1 module are listed as *ASN.1 type definitions by reference* in TTCN.
3. SDL signal definitions become *ASN.1 ASP type definitions*. As a consequence, if a signal is used in a test case, its corresponding TTCN constraint is stored in an *ASN.1 ASP constraint* table.

Very often, this mapping is too strict. For example, during test execution a tester may exchange both Abstract Service Primitives (ASPs) and Protocol Data Units (PDUs) with the system under test. If you want to store constraints as ASN.1 PDU constraints, you may use Define-TTCN-Signal-Mapping with parameter *PDU*. However, in this case *all* signals are considered to be PDUs. Moreover, this command does not apply to SDL sorts and ASN.1 data types.

In order to specify the correct for each different type of information, Autolink provides two commands in its configuration language. These commands start with either the keyword `ASP-TYPES` or `PDU-TYPES`. You can use them to declare single signals and sorts as ASPs and PDUs.

Example 283 ----------

```
ASP-TYPES
  "ICONreq" , "ICONconf", "IDATreq"
END

PDU-TYPES
  "pdu*"
END
```

In Example 282, three SDL signals, namely *ICONreq*, *ICONconf* and *IDATreq* are specified as ASPs. The second rule states that all signals and sorts whose name starts with "pdu" shall be considered to be PDUs. If constraints with corresponding types are used, they are stored as PDU constraints as well.

Stripping signal definitions

When it comes to the automatic generation of TTCN test suites, one of the drawbacks of SDL is that any data which is exchanged between a

system and its environment has to be encapsulated in signals. Especially, if your SDL specification makes use of ASN.1 data types, this restriction imposes a redundant embedding. On the other hand, the concept of signals does not exist in TTCN. Instead, common data values can be sent and received directly. For that reason, Autolink allows to strip signals.

Consider a signal type defined as *MDATreq(PDUType)*. If a signal of this type is used in a test case, say *MDATreq({ CC })*, then Autolink will generate a constraint of type *MDATreq*. However, what you may want to generate is a constraint of type *PDUType*, i.e. the signal should be stripped from its parameter.

Example 284 ––

```
STRIP-SIGNALS
  "MDATreq"
END

PDU-TYPES
  "PDUType"
END
```

––

Example 284 presents a short Autolink configuration statement that tells Autolink to unwrap the signal parameter when generating constraints that are related to signal *MDATreq* in the SDL specification.

Please note that signal stripping can only be applied to signals that have exactly one parameter! Moreover, the embedded parameter must be declared either as PDU or as ASP. If these conditions do not hold, Autolink refuses to strip the signal and issues a warning. If a signal can be stripped successfully, no declaration is generated for it in the TTCN declaration part, since it is not used in the constraints part.

Syntax and Semantics of the Autolink Configuration

Autolink Configuration

The definition of an Autolink configuration is started with the keyword `Define-Autolink-Configuration` and is terminated with `End`. It consists of an arbitrary sequence of five different kinds of statements:

Translation rules, test suite structure rules, ASP/PDU type rules, signal stripping rules and functions.

Example 285: Syntax of Autolink configuration

```
<Start>                ::= "Define-Autolink-Configuration"
                           <Configuration>
                           "End"
<Configuration>        ::= { <TransRule> | <TSStructureRule> |
                             <ASPTypesRule> | <PDUTypesRule> |
                             <StripSignalsRule> | <Function> }*
```

> **Note:**
>
> If you want to define both translation rules and test suite structure rules, you have to place them in the same configuration definition.

Rules and functions can be arbitrarily mixed in a configuration. There is no need to place rules on top of a file, nor do you have to write forward declarations for functions.

> **Note:**
>
> Autolink analyzes translation rules and test suite structure rules in the order they have been defined. As a consequence, the order of the definitions is crucial if several rules can be applied.

Translation Rules

Translation rules are evaluated whenever a constraint is created during test case generation.

A translation rule starts with the specification of the names of the signals to which it shall apply (denoted by `<AlternativeListOf-Terms>`).

Example 286: Syntax of translation rules

```
<TransRule>              ::= "TRANSLATE"
                             [ "SIGNAL" ] <AlternativeListOfTerms>
                               <TransRuleIf>* [ <TransRuleNoIf> ]
                              "END"
<TransRuleIf>            ::= "IF" <Conditions> "THEN"
                             <TransRuleNoIf> "END"
<TransRuleNoIf>          ::= { "CONSTRAINT" <TransRuleConstraint> |
                               "TESTSUITE" <TransRuleTestSuite> }*
<TransRuleConstraint>    ::= { "NAME" <Term> |
                               "PARS" <ParameterList1> |
                               "MATCH" <ParameterList1> }*
<TransRuleTestSuite>     ::= { "CONSTS" <ParameterList1> |
                               "PARS" <ParameterList2> }*
<ParameterList1>         ::= <Parameter1> { "," <Parameter1> }*
<Parameter1>             ::= "$" <Number> [ "=" <Term> ]
<ParameterList2>         ::= <Parameter2> { "," <Parameter2> }*
<Parameter2>             ::= "$" <Number> [ "=" <Term> ]
                                          [ "/" <Term> ]
```

As sketched in the example section, translations can be made dependent on one or more conditions. Hence, the body of a translation rule may consist of one or more statements embedded by `IF` ... `THEN` ... `END` constructs. The first group of statements whose preceding conditions are satisfied (or which do not have an `IF` statement at all) is evaluated. All subsequent definitions are ignored. If no conditions hold for a given signal, Autolink looks for another translation rule which fits the signal.

There are two groups of directives starting with either the keyword `CONSTRAINT` or `TESTSUITE`.

In the `CONSTRAINT` part you can specify the name (keyword `NAME`) and the formal parameters of a constraint (keyword `PARS`) for one or more given signals. Additionally, you can tell Autolink to replace signal parameter values by a TTCN matching mechanism (keyword `MATCH`). Please note that Autolink does not perform any checks concerning matching mechanisms at run-time. It simply handles it as a textual replacement.

In the `TESTSUITE` part, you can specify that parameter values of a signal are replaced by test suite parameters and constants. The declaration of constants is preceded by the keyword `CONSTS`, test suite parameter are introduced with `PARS`.

It is possible to declare a constraint parameter and a test suite constant/parameter for the same signal parameter. However, Autolink en-

sures that a signal parameter is not mapped onto a test suite constant and parameter at the same time.

There exist several default values that are used when an optional parameter is not specified:

- The default name of a constraint is defined by the term `'c' + $0`, i.e. the signal name is prefixed by a `'c'`.
- The default name of a constraint parameter is constructed by “Par” + <SignalNumber> (e.g. `Par3` for the third parameter).
- If a signal parameter is specified after `MATCH`, but no term is given, its values is replaced by ‘*’ in the constraint table.
- The default name of a test suite constant is “TestSuiteConst”.
- The default name of a test suite parameter is “TestSuitePar”.
- By default, there is no PICS/PIXIT reference.

If there are name clashes, test suite constants and parameters are treated similar to constraints and test steps. That means, if there are two constants with the same name but different values, they are distinguished by a sequence number.

Test Suite Structure Rules

Test suite structure rules are similar to translation rules. They share most of the basic concepts, for example terms, functions and conditions. However, while translation rules are applied during test case generation, TSS rules are evaluated when you save a test suite with the Save-Test-Suite command.

A test suite structure rule starts with the specification of the names of the test cases to which it shall apply (denoted by `<AlternativeListOfTerms>`).

Conditions can be used in the same way as in translation rules: The first `IN` statement whose preceding conditions are satisfied (or which is not embedded in an `IF` ... `THEN` ... `END` statement at all), is taken into account. All subsequent statements are ignored. If no conditions hold for a given test case/step, Autolink looks for another TSS rule that fits the test case/step.

Example 287: Syntax of test suite structure rules

```
<TSStructureRule>         ::= "PLACE" <AlternativeListOfTerms>
                              <TSStructureRuleIf>*
                              [ <TSStructureRuleNoIf> ]
                              "END"
<TSStructureRuleIf>       ::= "IF" <Conditions> "THEN"
                              <TSStructureRuleNoIf> "END"
<TSStructureRuleNoIf>     ::= "IN" <Term> { "/" <Term> }*
```

Declaring ASP and PDU Types

The rules to declare ASP and PDU types are evaluated when a TTCN test suite is saved on disk with the Save-Test-Suite command. Please note that each of the rules can only be defined once in an Autolink configuration. However, this is no restriction as you can specify an arbitrary number of signals and sorts in both rules.

Example 288: Syntax for declaring ASP and PDU types

```
<ASPTypesRule>       ::= "ASP-TYPES" <SequentialListOfTerms> "END"
<PDUTypesRule>       ::= "PDU-TYPES" <SequentialListOfTerms> "END"
```

Stripping Signals

Rules for stripping signals are evaluated closely coupled with the rules above for declaring ASP and PDU types. Autolink only strips a signal if its only parameter is declared as ASP or PDU. Autolink only accepts one stripping rule in a configuration.

Example 289: Syntax for stripping signals

```
<StripSignalsRule> ::= "STRIP-SIGNALS" <SequentialListOfTerms>
                       "END"
```

Functions

Functions are identified uniquely by their names. If there are two functions with exactly the same name, the one defined first is always evaluated.

Functions are visible globally, that is, they can be called by any constraint or test suite structure rule and other functions. References to

functions are resolved at run-time. If there is a call to an unknown function, the text `"FunctionXXXNotFound"` is returned.

Example 290: Syntax of functions

```
<Function> ::= "FUNCTION" <Identifier> <Mappings> "END"
<Mappings> ::= <Mapping> { "|" <Mapping> }*
<Mapping>  ::= <Conditions> ":" <Term>
```

A function body consists of a number of mapping rules separated by '|'. Mapping rules specify the possible return values of a function. A mapping is performed if its corresponding condition(s) hold. Mappings are evaluated from top to bottom. If the conditions of all mappings fail, a function returns the text `"NoConditionHoldsInFunctionXXX"`.

Function parameters can be accessed in the same way as signal parameters in a translation rule. For example, `$2` refers to the second parameter. In the context of functions, the reference `$0` denotes the name of the function. Since parameters do not have a name, but are referred to by their position instead, there is no need to declare them in the function header. If you try to access a parameter that has not been passed to the function, the missing parameter is replaced by the text `"ParOutOfRange"`.

> **Note:**
>
> In conditions, the existence of a particular parameter can be checked. For example, condition
>
> ```
> $4 == "ParOutOfRange"
> ```
>
> checks if four parameters have been passed to the function.

Basic Expressions

The only data type defined in the Autolink configuration language is **text**. Whether you refer to a signal parameter or call a function, the result of the operation is always a text.

Example 291: Syntax of basic expressions

```
<Term>                   ::= <Atom> { "+" <Atom> }*
<Atom>                   ::= "$" <Number> | "@" <Number> |
                             <Text> | <Identifier> |
                             <FunctionCall>
<FunctionCall>           ::= <Identifier>
                             "(" <SequentialListOfTerms> ")"
<SequentialListOfTerms>  ::= <Term> { "," <Term> }*
```

```
<AlternativeListOfTerms> ::= <Term> { "|" <Term> }*
<Conditions>             ::= <Condition> { "AND" <Condition> }*
<Condition>              ::= <Term> { "==" | "!=" } <Term> |
                             "TRUE"
```

Texts are constructed by atoms and terms. A single atom can be one of the following expressions, depending on the context in which the atom is used:

- A simple text (e.g. `"Request"`).
- An identifier (e.g. `Request`).
  Identifiers are treated as simple texts.
- A pattern (e.g. `"Sig*"`).
  Patterns can only be used in the header of constraint or test suite structure rules (for details see "Test Suite Structure Rules" on page 1465).
- A function call (e.g. `OpName($3)`).
  Function calls are not allowed in the header of constraint or test suite structure rules.
- A reference to a signal parameter (e.g. `$2`).
  References to signal parameters can only be used in the body of translation rules.
- A reference to a function parameter (e.g. `$2`).
  References to function parameters can only be used in the body of functions.
- A reference to an atom in the header of a constraint or test suite structure rule (e.g. `@2`).
  References to atoms can only be used in the body of constraint and test suite structure rules. Their application in combination with patterns is illustrated in "Test Suite Structure Rules" on page 1465.

Since an atom always evaluates to a text and a term is a concatenation of single atoms, you are allowed to use term expressions for the specification of:

- Constraint names
- Constraint formal parameter names
- Test suite parameter names
- Test suite constant names

- Test case/test step/test group names
- ...

Note:

The text obtained by referring to a signal parameter is identical to the output of the signal parameter value in ASN.1 format.

A condition checks whether two texts are equal (`==`) or unequal (`!=`). There is also a special condition `TRUE` that always evaluates to true.

Conditions can be combined by `AND`. Only if all conditions in a conjunction hold, the expression as a whole is true.

There is no `OR` operator for combination of conditions. However, due to the consecutive evaluation of rules (from top to bottom), this is not a restriction. For example, in a function body, simply place both `OR`-operands in two subsequent mappings.

Concurrent TTCN

In the 1996 version of ISO IS 9646-3, TTCN has been extended with mechanisms to specify test suites for distributed test systems. These extensions are known as *concurrent TTCN*. This section explains what will happen when you save your test suite in the concurrent TTCN format.

Declarations

In a distributed test environment, the test system is composed of a set of *Parallel Test Components (PTC)* which each handle one or more PCOs. The test system also includes one *Main Test Component (MTC)* which starts the PTCs and computes the final test verdict. The MTC may or may not control PCOs. The main and parallel test components exchange *Coordination Messages (CM)* through *Coordination Points (CP)*.

The collection of a number of test components and their connection through coordination points is called a *test configuration*. A test suite may contain more than one test configuration, and each test case has to be associated with a specific test configuration individually.

Coordination messages and corresponding constraints have to be declared similar to messages exchanged with the system under test, but in separate tables.

Dynamic behavior description

In concurrent TTCN, the Test Case Dynamic Behaviour table only describes the behavior of the main test component. The behavior of parallel test components is stored in Test Step Dynamic Behaviour tables. Obviously, the behavior tables of every test component contain only events observed at the PCOs and CPs attached to that test component.

Parallel test components are dynamically created by the main test component. This is done by the inclusion of `CREATE` statements in the test case behavior description. Similarly, the `DONE` event can be used in the test case description to check the termination of parallel test components. The final test case verdict is computed by the MTC from the verdicts returned implicitly by all PTCs before their termination.

Synchronization of test components

With concurrent TTCN, synchronization among test components becomes a necessity. Each test component only gets a partial view of the system under test and has no inherent knowledge of the state of the other test components. Therefore, the correct order of test events can only be established through the use of coordination messages. Consider the example shown in "Synchronizing Test Events with Conditions" on page 1444. If there are separate test components to control A, B and C, then C definitely has to wait for a coordination message before sending its `Request(B)` message to the SUT. If it does not, the test verdict entirely depends on the relative transmission time of the messages and the order of their handling by the SUT.

The Autolink implementation of concurrent TTCN

Autolink generates concurrent TTCN specific information only during the saving of a test suite. Therefore, it does not matter if concurrent TTCN is enabled during the generation or translation of test cases. You may generate your test cases, save the test suite in non-concurrent form, then turn on concurrence and save the test suite again.

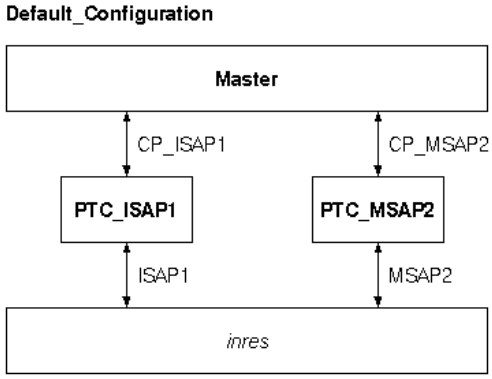


Figure 264: Test configuration for the inres system

Declarations

Autolink supports one kind of test architecture. From this generic architecture and the SDL specification of the system, a concrete test architecture is derived. The following declarations are generated automatically:

- One main test component called `Master`. The MTC does not control any PCOs.
- One parallel test component for each PCO. The name of the test component is `PTC_` + *Name of the PCO*.
- One coordination point between the MTC and each PTC. The name of the coordination point is `CP_` + *Name of the PCO*.
- One test configuration called `Default_Configuration`, which contains all test components and their connections with PCOs and CPs.

Figure 264 shows the test configuration which is generated for the *inres* System. In addition, the following declarations are generated:

- One ASN.1 CM Type Definition. The name of the coordination message is `CM` and its definition is `SEQUENCE {message PrintableString}`.

- Two ASN.1 CM Constraint Declarations, called `Proceed_Indication` and `Ready_Indication`. Both constraints define values for the coordination message `CM`.

`CM`, `Proceed_Indication` and `Ready_Indication` are used for coordination messages between the main and parallel test components. These messages are generated automatically (see Synchronization below).

> **Note:**
> The test architecture and resulting default test configuration can not be changed within Autolink. The coordination message and corresponding constraints are not changeable neither.

Test Case Dynamic Behaviour

Test Case Name : mi_synch2
Group :
Purpose :
Configuration : Default_Configuration
Default : OtherwiseFail
Comments :
Selection Ref :
Description :

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		CREATE(PTC_ISAP1:mi_synch2_PTC_ISAP1)		(PASS)	
2		CREATE(PTC_MSAP2:mi_synch2_PTC_MSAP2)			
3		+Synchronization			
4		? DONE(PTC_ISAP1,PTC_MSAP2)		R	

Detailed Comments :

Figure 265: Sample test case description for a main test component

Dynamic behavior description

Autolink splits the internal test case description into separate trees for every test component. The Test Case Dynamic Behaviour table describes the behavior of the main test component; its name is equal to the name of the original MSC test description. Since the MTC does not control any PCOs, it only contains `CREATE` statements for every PTC at the beginning and a `DONE` event at the end. It may also contain attached synchronization test steps in between. Figure 265 shows a sample test case description. The `(PASS)` verdict on line 1 initializes the result variable `R`. At the end of the test case, `R` contains the final test verdict.

Each parallel test component gets a Test Step Dynamic Behaviour table of its own. The name of the test step is *Test case name* + _ + *Test component name*.

Synchronization

Coordination messages are automatically generated by Autolink wherever a condition appears in the MSC test description. As a consequence of the test architecture used by Autolink, synchronization is done via the main test component. Here is a description of the algorithm:

1. For each MSC environment instance connected to a condition: Send a coordination message `CM` with constraint `Ready_Indication` to the MTC.
2. For each MSC environment instance connected to a condition which has a send event immediately following the condition: Receive a coordination message `CM` with constraint `Proceed_Indication` from the MTC.

Test Step Dynamic Behaviour

Test Step Name : Synchronization
Group :
Objective :
Default : OtherwiseFail
Comments :
Description :

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		CP_MSAP2 ? CM	Ready_Indication		
2		CP_ISAP1 ? CM	Ready_Indication		
3		CP_ISAP1 ! CM	Proceed_Indication		
4		CP_ISAP1 ? CM	Ready_Indication		
5		CP_MSAP2 ? CM	Ready_Indication		
6		CP_ISAP1 ! CM	Proceed_Indication		

Detailed Comments :

Figure 266: Synchronization test step for a main test component

From the viewpoint of a parallel test component, this means that whenever it reaches a synchronization point, it sends a `Ready_Indication` message to the MTC. If the event immediately following the synchronization point is a send event, then the PTC first waits for a `Proceed_Indication` message, which it receives from the MTC. All coordination events are directly included in the dynamic behavior description of the PTC.

When it reaches a synchronization point, the main test component waits for `Ready_Indication` messages from every PTC involved in the synchronization. Afterwards, it sends `Proceed_Indication` messages to all PTCs which are about to send a message to the system under test. Since the reception of coordination messages from different PTCs can create a lot of alternative paths, all synchronization events for the MTC are put into test steps. Figure 266 shows an example of a synchronization test step for an MTC.

> **Note:**
>
> Coordination messages can not be specified manually in the MSC test description. There are two reasons for this: First, the Explorer and Autolink can not handle messages drawn between environment instances. Second, all instances in the MSC have to relate to a channel in the SDL specification. Since the main test component has no connection with the system under test, it is not possible to add an MTC instance to the MSC.

Caveats

In order to streamline test suites and enhance their readability, Autolink automatically merges test steps which contain identical behavior descriptions. Furthermore, empty test steps are removed.

If your MSC test descriptions contain MSC references and concurrent TTCN is used to save the test suite, then the test step streamlining of Autolink may lead to unexpected results: For example, test steps for parallel test components may be renamed unexpectedly. Within test case and test step behavior descriptions, expected attachments of test steps may be missing. Nevertheless, these test suites are still correct and correspond to the original test descriptions.

Test Suite Timers

In this section, details regarding the declaration and use of test suite timers in test case MSCs is discussed.

Timer declarations

As explained in "Using Timers" on page 1438, the recommended method for using test suite timers with Autolink is to explicitly declare all timers which appear in the test suite with Define-Timer-Declaration before test generation is started. Nevertheless, it is possible to have a mix-

ture of implicit and explicit declaration. Below, the rules are listed which Autolink applies when creating or updating timer declarations. In any case, the syntactical correctness of the timer name is not checked.

Creation of a new explicit timer declaration

- Autolink checks if the duration is an integer value or a syntactically correct test suite parameter. If it is neither, then a warning is displayed and the duration field remains empty. If it is a test suite parameter, a test suite parameter declaration is created in addition to the timer declaration.
- No declaration is created if the unit is not valid.

Note:

The only possibility to specify an empty duration field on purpose is to use an invalid string, e.g. a digit followed by a character.

Creation of a new implicit timer declaration

- Autolink checks if the parameter field of the timer set symbol ends with a valid unit, which means that there must be a whitespace character followed by either *ps*, *ns*, *us*, *ms*, *s* or *min*. If this is the case, then the value of the unit field is set and the rest of the string is considered to be the duration. If no valid unit can be found, then the whole parameter field is considered to be the duration.
- Autolink checks if the duration is an integer value or a syntactically correct test suite parameter. If it is neither, then a warning is displayed and the duration field remains empty. If it is a test suite parameter, a test suite parameter declaration is created in addition to the timer declaration.

Note:

It is the responsibility of the test designer to make sure that either an explicit timer declaration exists or that an implicit declaration is correct. Autolink does not guarantee that a test suite which contains implicit timer declarations passes a TTCN syntax check.

Update of an explicit declaration with an explicit one

An existing explicit declaration can not be updated. A warning is displayed and the new declaration is ignored.

The only way to remove existing timer declarations is to use the Reset command.

Update of an explicit declaration with an implicit one

An existing explicit declaration can not be updated. However, Autolink checks if the unit of the implicit declaration matches the unit of the existing explicit declaration. If it does not match, a warning is displayed.

Update of an implicit declaration with an explicit one

Autolink checks if the unit of the new declaration is valid. If it is not, the existing implicit declaration is kept. If the unit is valid, the implicit declaration is replaced by the explicit one.

Update of an implicit declaration with an implicit one

- Autolink compares the unit of the existing declaration with the unit of the new declaration:
  - if the existing unit is valid and the new one is invalid, then the existing one is kept;
  - if the existing and the new unit are identical, then the unit is not changed;
  - in all other cases, the unit field is cleared; this will result in a syntactically incorrect TTCN test suite.
- If the duration field of the existing declaration is empty or an integer value and the new duration is a test suite parameter, then the test suite parameter replaces the existing value.

Timer pitfalls

Timers in a test suite are declared globally. During test execution, each participating test component receives a complete set of timers which it can use independently. With respect to the readability of a test suite, the number of timer declarations should be minimized. This can be accomplished by declaring a minimal set of timers and reusing them in different test case MSCs.

If concurrent TTCN is enabled, identically named timers may also be used on different instances in the same MSC, because in the resulting TTCN test case, each MSC instance is handled by a different test component. However, if such an MSC is used in a non-concurrent context,

then Autolink may produce unexpected test sequences and care should be taken.

Timer optimization

If concurrent TTCN output is enabled with Define-Concurrent-TTCN, then the dynamic behavior descriptions are optimized with regard to the placement of timer operations. If a timer *START* operation is followed immediately by a send event, then the *START* operation is placed on the same line as the send event. Correspondingly, if a timer *CANCEL* operation follows a receive event, then the *CANCEL* is moved up to the line with the receive event. As an example, if the test sequence according to the test case MSC is

```
START T1
  A ! someSignal
    A ? anotherSignal
      CANCEL T1
```

then Autolink optimizes this and generates the following test sequence:

```
A ! someSignal  START T1
  A ? anotherSignal  CANCEL T1
```

Chapter

36 *Adaptation of Generated Code*

This chapter contains information about making an adaptation of the generated code to make it communicate with the generated code.

The first part of this chapter describes the interface to be used when writing the adaptation, the GCI Interface. The different parts of the interface are explained in detail with recommendations and examples of use.

The second part of this chapter is a C implementation of GCI, this can be used as a reference documentation.

The third part describes the EGci (extended GCI) value construction and functions.

The fourth section introduces the Adaptation Framework that can be used to implement GCI functions in a straightforward way with ready-made low-level protocol implementations such as TCP/IP.

The fifth and final section describes the means and measures to complete the adaptation. It describes the requirements on the adaptation in terms of the necessary functions to implement.

The GCI Interface

The Generic Compiler Interpreter (GCI) interface standardizes the communication between a TTCN component supplied by a vendor and other test system components supplied by the customer, together forming a MOT, Method Of Test.

The GCI interface focuses on what an ATS needs in order to execute in term of functionality, and on what is needed in order to integrate TTCN into a larger system. This chapter contains a natural language description of the interface and a GCI C Reference.

The GCI Interface Model

The main purpose of the GCI interface is to separate TTCN behavior from protocol and test equipment specific code. The GCI interface shall be a standardized set of functions. Communication shall be done by calling functions, passing arguments to the functions and return values from the functions, and in no other way. The interface is bidirectional which means that both parties (the TTCN Runtime Behavior and the Test Adaptation) must provide services to each other. The TTCN Runtime Behavior shall at least provide services for handling values and managing tests (evaluating test cases etc.) and the Test Adaptation shall provide the protocol/test equipment specific parts of an executable (send, snapshot, timer functions etc.). The implementation of the functions in the TTCN Runtime Behavior may only depend on the ATS and 9646-3, no extra information shall be needed to implement them.

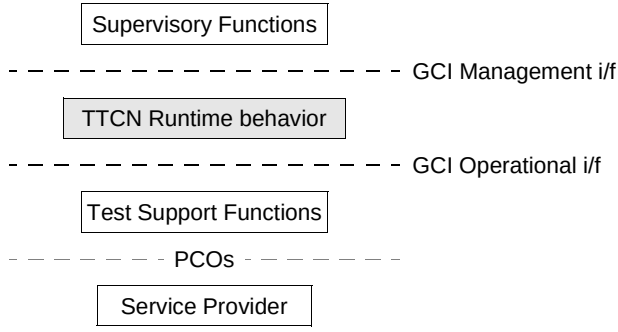


Figure 267: The GCI model

Informal Description of the Test Run Model

This section contains an informal description of how a test run might look like using the GCI interface. A test run is defined as a complete test session, where a selection of test cases are run to ensure a specific behavior of the IUT.

A test run begins when the user decides to run a test case or a collection of test cases. He will then use the Supervisory functions to start test cases and monitor their verdicts. The TTCN Runtime Behavior then executes TTCN, occasionally using the operational interface (send, snapshot, etc.) to gather information or to request some service. The TTCN Runtime Behavior handles verdicts internally during a test case and returns the verdict at the end of a test case.

Before using the TTCN Runtime Behavior, it must be initialized. After that the user chooses which test case to run using the supervisory functions. The test case returns a verdict which the user can use to form reports or stop test runs. The TTCN Runtime Behavior or the GCI interface does not impose any restrictions on this part. Any number of test cases can be run and each is commanded using the Supervisory functions.

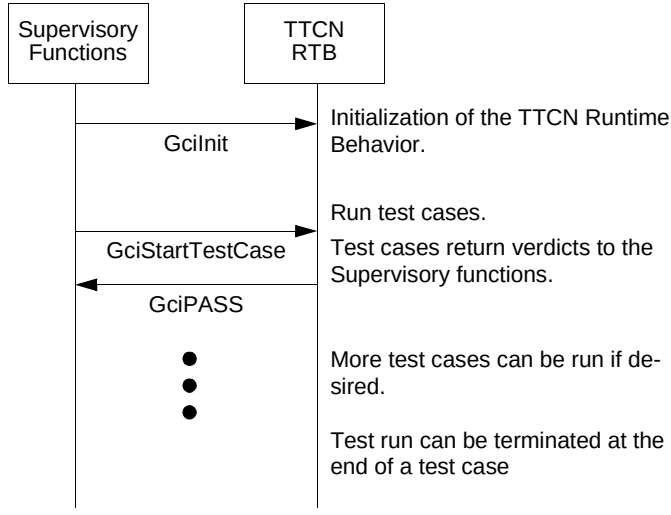


Figure 268: Start of test run

The test case at some point will try to send a message on some PCO. The TTCN Runtime Behavior then passes information to the Test Adaptation about the message to send and on which PCO to send it on. It is the responsibility of the Test Adaptation to properly encode the message and actually send it on some media (e.g. sockets, screen, printer port, pipe). Note that control is in the Test Adaptation until it returns to the TTCN Runtime Behavior. When control returns to the TTCN Runtime Behavior it assumes that the message was sent correctly and continues execution of the test case.

Eventually the test case will have emptied its possibilities to act and needs input from the environment. It therefore passes control to the Test Adaptation in order to take a snapshot of the IUT. Within the snapshot, the Test Adaptation then checks all PCO's for incoming messages and all timers for time-outs. If a message has arrived on a PCO, the Test Adaptation must decode the message and translate it into a proper GCI value. If a timer has timed out, the Test Adaptation must record which timer. The Test Adaptation acts as a filter between the IUT and the TTCN Runtime Behavior. Note that the actual reception of a message or time-out could be handled somewhere else (e.g. in an interrupt routine), only the official communication that the event has taken place must be done in SNAPSHOT.

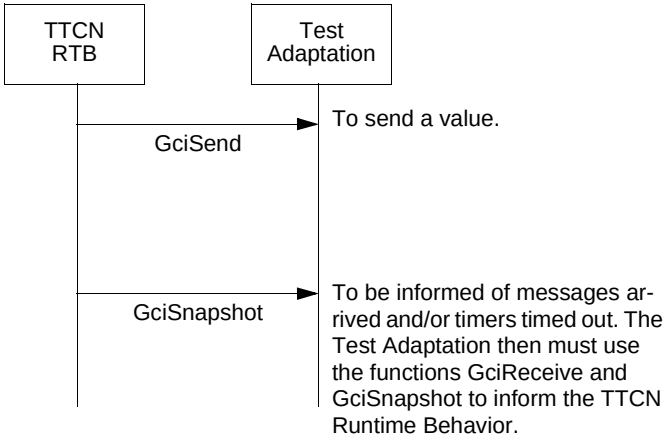


Figure 269: Send and Snapshot

Preliminary verdicts may be set during the test execution and when a final verdict is set, the test case returns to the Test Adaptation immediately. The Test Adaptation then has freedom to decide to continue the test run or not. A final verdict has meaning only in the context of the current test purpose. The meaning of a final verdict is not specified in the context of an entire collection of test cases. For example, when doing regression tests, all test cases should be run regardless of how many failed in order to get a complete picture of the status of the IUT.

Which Does What?

The table below summarizes the responsibilities of the two GCI parties, the TTCN Runtime Behavior and the Test Adaptation. It is meant to describe the model and basic assumptions being made.

TTCN Runtime Behavior	**Test Adaptation**
SEND: Builds the object to be sent and requests the actual sending from the Test Adaptation.	SEND: Receives the object that shall be sent and transfers this to the IUT. This might include encoding in some form.
RECEIVE: When a value is put on a PCO queue by the Test Adaptation, it remains there until a matching receive line is found in TTCN.	RECEIVE: Builds the received object (from the IUT) into a GCI value, which may include some form of decoding, and notifies the TTCN Runtime Behavior that the message has arrived.
Sets the verdict (based on the test case).	Treats the verdict (i.e. decides if the test run should continue or not).
Impl. of value representation.	Encoders/decoders.
Uses the value repr.	Uses the value repr.
Provides test cases. The test cases are sensitive to test case selection references.	Determines which test cases to run and in which order.
Uses send on messages.	Provides the send functionality.

TTCN Runtime Behavior	Test Adaptation
Uses SNAPSHOT to fix a view of the status of the IUT. Expects all changes in system status to be recorded in SNAPSHOT.	Reads system status in SNAPSHOT and records this through GCI interface for the TTCN Runtime Behavior.
Uses the LOG function to log TTCN Runtime Behavior.	Provides the LOG functionality, and also decides on what level logging should be done.

Case Studies

This section contains case studies of a send and a receive. For the case studies, we use an example PDU shown in Figure 270.

TTCN PDU Type Definition		
PDU Name: pdu1		
Field Name	**Field Type**	**Comment**
a	INTEGER	w value 19
b	BOOLEAN	w value FALSE
Comment: Example pdu for GCI case studies		

Figure 270: Example table

Case Study: SEND

Nr	Lbl	Statement Line	Cref	Comment
		`L ! CR`	`CR_c`	

The semantic of the TTCN send statement is as follows:

1. Build the SendObject using the constraint reference.
2. Execute the assignments.
3. Send the SendObject on the PCO.
4. Execute the timer operations.
5. LOG, at least the PCO and the SendObject must be logged.

The SendObject above is a temporary object containing the object to be sent. All steps above are initiated by the TTCN Runtime Behavior.

Steps 3 to 5 require communication with the environment (IUT). The TTCN Runtime Behavior will build a SendObject using the constraint CR_c above. Then it will call the Test Adaptation function Send. By doing this, the TTCN Runtime Behavior has ensured that the message gets sent on whatever PCO was stated. The Test Adaptation function Send then encodes the message using the transfer syntax of the protocol and then sends the message on the medium that represents the PCO. The arguments passed *to* Send is the PCO and the message in the GCI representation, and the *side effect* of Send is to send the message in protocol representation on the PCO in test system representation. Note that the encoding and sending on a PCO might very well be represented as a function call.

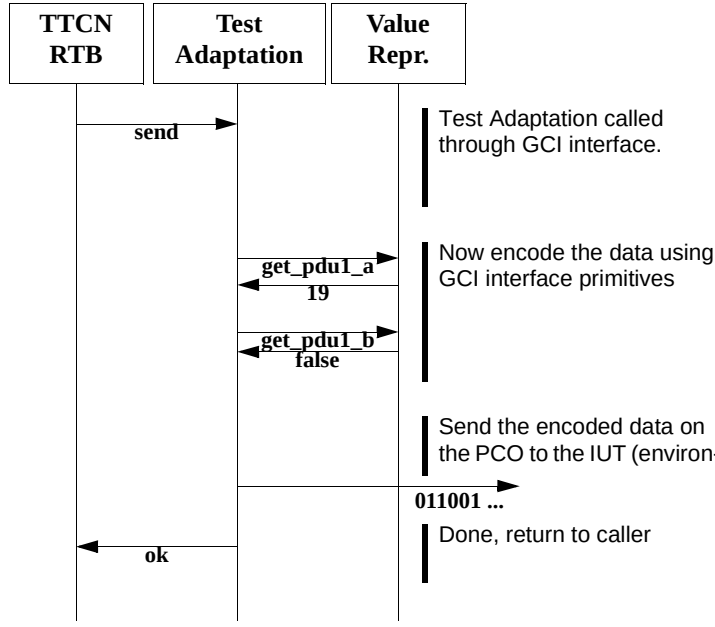


Figure 271: The send event

Case Study: RECEIVE

Nr	Lbl	Statement Line	Cref	Comment
		L ? CC	CC_c	

The receive is a little more complicated than the send. Send is initiated by the TTCN Runtime Behavior while receive is an internal event in the TTCN Runtime Behavior. The actual reception of messages is done in the Test Adaptation and these actions are communicated to the TTCN Runtime Behavior in SNAPSHOT. Note that there may be a difference between the reception of a message and the notification to the TTCN Runtime Behavior that the message has arrived. The TTCN Runtime Behavior calls the function SNAPSHOT and when it returns, the TTCN Runtime Behavior expects all time-outs to have been recorded and all received messages to be put on the correct PCO queues in the correct format. In this case we study the message event only and not the time-out.

> A message has been received from the IUT (not necessarily in SNAPSHOT) and the TTCN Runtime Behavior must be told that the message has been received. This shall be done in SNAPSHOT using the GCI function provided. The message and the PCO must be presented in a representation understood by the TTCN Runtime Behavior. Note that the message probably needs to be translated into the GCI value representation somewhere but again not necessarily in SNAPSHOT.

If SNAPSHOT does what is stated above, the TTCN Runtime Behavior will do the following on the receive line in the test case:

> The PCO queue is checked and if there is a message there, that message is matched against the constraint reference. If the message matches, it is removed from the PCO queue and the receive statement is considered to be TRUE, otherwise, the next alternative is checked.

In Figure 272, we assume that decoding is done in SNAPSHOT. The message is decoded using the value manipulating primitives of the GCI interface. When the message has been decoded into something known to the TTCN Runtime Behavior, it is easy to assign it to a PCO using the GCI function receive.

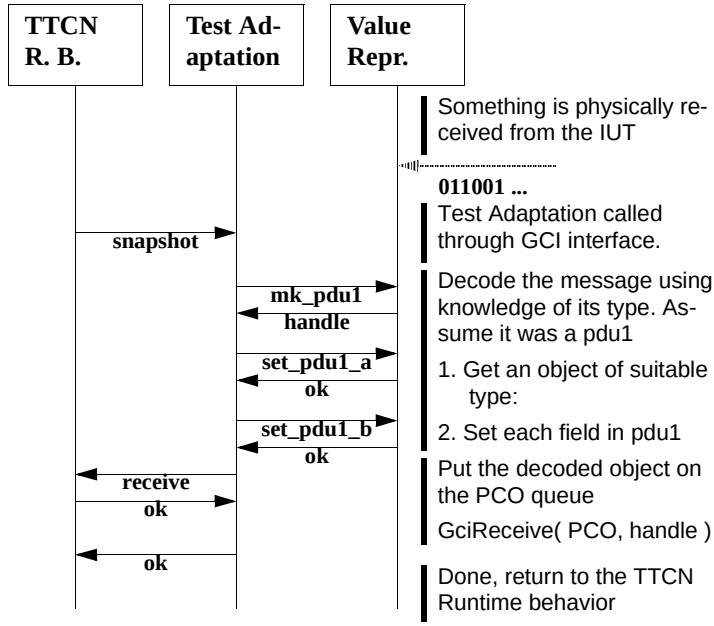


Figure 272: The receive event

Methods Used

This section contains general thoughts on methods used for designing the GCI interface. Choices have been made such as to provide reasonable trade-offs between speed and ease of use. An efficient implementation in C should be possible with the choices made.

Identifying Global Objects in TTCN/ASN.1

One problem is how to identify global objects in TTCN, PCOs, CPs, ASPs etc. There are a number of requirements that should be met:

1. Execution time:

   The TTCN behavior, operational and value interface functions are likely to be called many times in a time sensitive context which requires them to be fast. The management functions does not have this requirement.

2. Compilation time:

   The Test Adaptation and the TTCN Runtime Behavior are likely to be used together with a graphical interface (GUI) or integrated into a larger system. This integration should be possible without re-compiling the GUI or the larger system every time a change has been made in the ATS. This requires that dependencies between the ATS and the management interface implementation must be kept to a minimum.

There are at least three alternative solutions to be considered:

1. Straight name reference using target language semantics, i.e start test case “TC1” by writing the function call “TC1()” in C, and the value of the variable “TCVAR1” is expressed simply as TCVAR1

2. Assign an integer to each object. Symbolic names could then be used. The examples from above would then be:

```
#define TC1 192
#define TCVAR1 1211
Verdict = GciStartTestCase( TC1 );
Display( GciGetValue( TCVAR1 ));
```

3. Use string references.

```
Verdict = GciStartTestCase( TC1 );
Display( GciGetValue( TCVAR1 ));
```

Solution 1 gives the fastest execution but is very unsuitable for integration into a larger application. Solution 2 is at least possible to integrate into a GUI, and almost as fast as solution 1. Solution 3 is excellent for integration but gives slow execution.

In the GCI interface, we choose to use solution 3 (string references) in the management functions and solution 2 in the TTCN behavior functions. It would then be very easy to integrate an ETS into an already existing test system since the interface is string based while we still maintain speed in the execution of test cases.

A similar problem is how to identify fields in structured types. The simplest solution is to address them using integers field 0, field 1, etc., but then the Test Adaptation writer would get very little support from the compiler. From the Test Adaptation writers point of view, he might want to address the fields using names and also get type checking aid from the compiler. The Test Adaptation writer also wants to write generic functions for encode/decode. The GCI interface supports both

views, one view where substructures are identified by numbers, and one which in effect use name addressing.

Test Suite Operation Definitions

The TTCN Runtime Behavior requires that all test suite operations are defined as functions at run-time. Arguments to and return values from the test suite operation functions shall use the GciValue* format. The name of the test suite operation function shall be the same as the name of the test suite operation in TTCN.

Logging

The log will provide a view of a test run. Most things happening in the system will need to be logged. We have identified a number of requirements on the log functionality:

1. It should be possible to only log every event visible to the TTCN Runtime Behavior. I.e. the sending of a message should leave a note in the log while the internal behavior of the TTCN Runtime Behavior should not.
2. For debug purposes, it must be possible to report every event in the system.
3. The granularity of the log must be easy to change by the Test Adaptation writer. It should be possible to be quite specific when logging, i.e. log only sends and receives.

The method that the GCI interface suggests is massive logging. Everything is logged at any time and every type of event (send, receive etc.) is assigned a number. That way it is easy for the Test Adaptation writer to write a log function that only logs interesting log messages.

Value Representation

The GCI interface must provide a stable way of manipulating values. This is most important for the Test Adaptation writer to be able to access and build values in his encode and decode functions. The GCI interface proposes not to specify the value representation but rather the **methods** which can be used on it. The GCI interface specifies what the Test Adaptation writer is able to do with values. The reason for this is that most vendors supporting the GCI interface would like to have their own value representation within their TTCN Runtime Behavior, and the

representation used within TTCN has different requirements than the representation needed for the Test Adaptation.

Introduction to the GCI Interface

This section contains the description of the GCI interface in natural language. The interface functionality is sorted by responsibility. The management, behavior and value interfaces are interfaces to the TTCN Runtime Behavior. The operational interface is the interface to the Test Adaptation.

Management Interface

The management interface consists of those functions necessary for initiating and managing test runs. They provide an API to the ATS. These functions are used by the Supervisory functions to govern test runs.

Initiation of the TTCN Runtime Behavior

Must be called before the TTCN Runtime Behavior is used in any way. It will read test suite variables (using primitives in the operational i/f), set up test case selection references, test suite constants and any other initialization necessary. It should only be called once.

Start Test Case

Shall run a test case according to the dynamic behavior in 9646-3. Input shall be a test case name or test case group name. The test case or all the test cases in the test group shall then be run in the order which they were listed in the ATS. If the selection reference of a test case or test case group is false, that test case or test case group shall not be executed. Verdict shall be set and communicated back to the caller. In case several test cases have been run, the verdict shall be the most negative of the verdicts from each test case (if one test case is FAIL, the verdict would be FAIL, if all are PASS, the verdict would be PASS).

Start Test Component

This function works just line the function to start test cases, but starts test components for example when concurrent TTCN is used.

Test Case List

A list of names of test cases.

Test Case Group List

A list of names of test case groups.

Number of Timers and PCOs

Two functions are needed to return information about the amount of timers and PCOs in the system.

Information About Timers and PCOs

Help functions are needed to get some extra information about timers and PCOs. (For example name and type.)

Configuration Information

A set of functions is needed to retrieve information about configurations when running concurrent TTCN.

Accessing TTCN Values

Values of TTCN (and ASN.1) objects must be accessible from the Test Adaptation. A general access function will be provided. Information passed shall be an object name and the information passed back shall be the value of the object. Valid objects shall be test suite parameters, test case selection expressions, test suite constants, test suite variables and test case variables.

Behavior Interface

The behavior interface consists of functions used to notify the TTCN Runtime Behavior of events in the IUT. The functions are a formalism of what the TTCN Runtime Behavior needs to know about the status of the IUT. The functions are implemented in the TTCN Runtime Behavior and must be used in Snapshot.

Receive

Whenever a message has been received and decoded it must be passed to the TTCN Runtime Behavior in SNAPSHOT in order for it to match it with the constraint. The information passed shall be the PCO descriptor and the value. Note that the value passed in receive must be a GciValue*.

Time Out

Whenever a timer has timed out the TTCN Runtime Behavior must be notified of it in SNAPSHOT. The information passed shall be the timer descriptor.

Done

Whenever a test component has terminated its execution the TTCN Runtime Behavior needs to know about this. The information passed is the descriptor for the given test component.

Operational Interface

The operational interface consists of those primitives necessary for the TTCN Runtime Behavior to implement TTCN, i.e. Send, Snapshot, StartTimer etc. This part of GCI can be compared to POSIX. TTCN Runtime Behavior requires that the functions are defined.

Test Suite Parameters

The TTCN Runtime Behavior shall call the Test Adaptation once for each test suite parameter during the initialization of the TTCN Runtime Behavior. Information passed shall be a handle for the value, the parameter name and the PICS/PIXIT reference.

Test Suite Operations

Each test suite operation must be defined in the Test Adaptation. The name of the test suite operation function shall be the same as in the ATS, and the order of the parameters shall also be the same.

Create

When a test component needs to be created this function will be called from the TTCN Runtime Behavior.

Configuration

When different tests are run, different configurations might be needed. This function is called from the TTCN Runtime behavior to set up a configuration before the test goes on.

Snapshot

The status of the IUT must be read. Events in the IUT must be translated for the TTCN Runtime Behavior to be aware of them. The communicating interface between the TTCN Runtime Behavior and the IUT (environment) is made up of messages that has arrived and timers that have timed out. Therefore there are only two requirements on SNAPSHOT: 1) any received messages shall be recorded and 2) any timers that have timed out shall be recorded. The recording shall be done by using the provided GCI functions described in the Behavior i/f.

Send

This function is used by the TTCN Runtime Behavior to ensure that messages get sent. Send probably involves encoding and physical sending but might as well translate to encoding of some parts of the message and use of a lower layer service. The information passed to the function shall be a PCO descriptor and the value to be sent.

Start Timer

When the TTCN Runtime Behavior needs to start a timer. Information passed shall be the timer descriptor, the integer timeout duration and the timer unit.

Read Timer

Shall pass the current timer value back to the caller. Information passed to the function shall be a timer descriptor.

Cancel Timer

Shall stop a timer identified by a timer descriptor.

Log

Shall log events on an appropriate format. An integer specifying log message type and a log string shall be passed as parameters.

Value Interface

The interface to values is a simple API in which the user is allowed to build and access GciValues in an ordered way. The actual values used are not specified, only the methods that can be used on them are included. This way will allow vendors to have their own value representation within their TTCN Runtime Behavior and still conform to GCI.

Three types of value primitives are needed. Primitives to access structured values:

- Primitives to build and access components of structured values
- Primitives to find and set the type of values
- Primitives to convert base values to and from the value domain of the target language

Value primitives can be further divided in the groups:

- SEQUENCE values, which include SET values and also TTCN ASP, PDU and Struct types
- SEQUENCE OF values, which include SET OF values
- CHOICE values
- OBJECT IDENTIFIER values
- BASE values, integers, booleans, strings and reals

Please note that SET and SET OF values are treated as SEQUENCE and SEQUENCE OF values because there is no semantic difference at this level of abstraction. The unordered behavior of SETs is considered a decoder problem and left to the decoder writer.

GCI C Code Reference

This section is a code description of GCI. It describes what each function does and why it is used. Some of the functions must be implemented by you in your adaptation. See section "Completing the Adaptation" on page 1546.

Predefined Types

These are the GCI value types with their respective value set:

```
GciStatus = { GciNotOk, GciOk }

GciVerdict = { GciFail, GciInConc, GciPass, GciNone}

GciTimeUnit ={ Gcips, Gcins, Gcius, Gcims, Gcis, Gcimin }

GciTimerStatus = { GCIRUNNING, GCISTOPPED, GCIEXPIRED }

GciPCORole = { GciUT, GciLT }

GciPCOID = INTEGER

GciTimerID = INTEGER

GciPosition = <internal position value>

GciSnapshotStatus = { GciSnapshotEvent,
GciSnapshotNoEvent, GciSnapshotNotOk }
```

These are structured GCI value types with their respective members:

The general communication address type

```
typedef struct GciAddress {
  int     type;        /* Address type 0 - 100 is
                          reserved by IBM Rational */
  char*   buffer;      /* The address stored in a
                          buffer */
} GciAddress;
```

The general time value

```
typedef struct GciTime {
  unsigned long time_val;
```

```
  GciTimeUnit   unit;
} GciTime;
```

Management Interface

The management interface consists of the functions necessary for initiating and managing test runs. They provide an API to the TTCN runtime behavior.

`GciStatus GciInit()`

Initiates the TTCN Runtime Behavior. Must be called before any test cases are started.

`GciStatus GciExit()`

Shuts down the TTCN Runtime Behavior system. Usually called last before closing a test session.

`GciVerdict GciStartTestCase( const char* TCorTGName )`

Calling this function with a test case name will start and run the given test case. The verdict returned is the verdict set by the test case.

If the function is called with a test group name, it will start and run the given test group. The verdict returned is the product of the verdicts set in the test cases or test groups contained in the given test group. The individual verdicts from the different test cases or test groups have to be extracted from the log. The verdict algorithm for test groups is defined as follows:

1. The verdict is PASS if the verdict of every test case is PASS.
2. The verdict is INCONC if the verdict of at least one test case is INCONC and all others are PASS.
3. The verdict is FAIL if the verdict of at least one test case is FAIL.

If the given name is not a valid test case nor test group, the function will return `GciNotOk` and log the following message:

```
No such Test Case or Test Group as <name> to run!
```

GciVerdict GciStartTestComponent(char* TSName, GciValue args)**

This function works just like GciStartTestCase but is used to start a parallel test component (PTC) when using concurrent TTCN. TSName is the name of the Test Step to run in the PTC.

GciTCList* GciGetTestCaseList()

This function returns a list of all test case names (including test cases in test groups).

GciTCList GciGetTestCaseGroupList()

This function returns a list of all test group names (including test groups in test groups). List of character strings.

GciValue* GciGetValue(char* TTCNObjectName)

This function returns the TTCN value object given the object's name. If the object name does not exist, the function will return NULL and log the following message:

```
GciGetValue: Could not find <name>! NULL returned
```

If the object name is a TTCN name, but not an object (variable, constant, etc...) the function will return NULL and log the following message:

```
GciGetValue: <name> not an instance! NULL returned
```

int GciGetNoOfTimers( )

Returns the number of timers in the system.

int GciGetTimer(int index)

Returns the identifier of the given timer index.

char* GciGetTimerName(int timer)

Returns the name of the given timer.

int GciGetTimerIndex(int desc)

Returns the index of the given timer descriptor.

int GciGetNoOfPCOs( )

Returns the number of PCOs in the system.

int GciGetPCO(int index)

Returns the descriptor of the given PCO index.

char* GciGetPCOName(int pco)

Returns the name of the given PCO.

int GciGetPCOIndex(int desc)

Returns the index of the given PCO descriptor.

int GciGetPCOType(pco)

Returns the type of a given PCO.

GciStatus GciGetPCORole(int pco, GciPCORole *role)

This function finds the role of PCO from its ID. Returns GciOk and assigns the role value to '`role`' when successful. Returns GciNotOk when the function fails.

GciStatus GciClearAllPCOSAndCPS(void)

This function clear all PCOs and CPs queues. There is no need to call this function before starting a Test Case as the queues are cleared automatically.

int GciGetNoOfComponents(GciConf conf)

Returns the number of components in the given configuration.

GciComponent* GciGetComponent(GciConf conf, int index)

Returns the indexed component in the given configuration.

char* GciGetComponentName(GciComponent* conp)

Returns the name of the given component.

int GciGetComponentType(GciComponent* conp)

Returns the type of the given component (GciMTC or GciPTC).

int GciGetComponentDescriptor(GciComponent* conp)

Returns the descriptor of the given component.

char GciGetComponentCPs(GciComponent* conp)**

Returns the CPs used by the given component.

char GciGetComponentPCOs(GciComponent* conp)**

Returns the PCOs used by the given component.

Behavior Interface

The behavior interface consists of the functions used to notify the TTCN Runtime Behavior of events in the IUT. The functions are a formalism of what the TTCN Runtime Behavior needs to know about the status of the IUT. The functions are implemented in the TTCN Runtime Behavior and must be used in Snapshot.

GciStatus GciTimeout(int timerd)

This function will change the status of the internal timer indexed by the timer descriptor `timerd`. The timer will be marked as timed out.

GciStatus GciReceive(int pcod, GciValue* value)

This function will insert the received object into the internal PCO queue indexed by the PCO descriptor `pcod`.

Operational Interface

The operational interface consists of the functions necessary for the TTCN Runtime Behavior to communicate with the IUT, i.e. the dependencies on the protocol and test environment. It implements functionality for Send, Snapshot, StartTimer, etc. The TTCN Runtime Behavior requires that the functions are defined in the adaptation.

GciValue* GciReadTSPar(char* name, char* pRef)

The user has to implement how to read the test suite parameters. This code will be called one time for each test suite parameter in the system. Here `name` is the name of the test suite parameter and `pRef` is the PICS/PIXIT field (ex. file). The value returned by this function must be a pointer to a valid `GciValue` structure which can be successfully used by the TTCN runtime behavior. If for any reason this could not be done, NULL should be returned.

GciStatus GciConfig(GciConf conf)

When using concurrent TTCN the user is responsible for creating the configurations to be used for a given test. The creation of those configurations is done in this function using a set of help functions in the management interface to traverse the information.

GciStatus GciSnapshot()

This is a very important function that the user has to implement. This function is called from the run-time behavior when it needs input from the test environment (timers and/or PCO's).

- Communication lines (PCOs) must be checked. If something has been received, we must decode the message, build a receive object and insert the object into the correct, internal PCO queue by using the `GciReceive` function with appropriate PCO descriptor. The GCI Value Interface is used to build the object(s).
- Timers must be checked and their status must be reported to the TTCN run-time system. If the user uses the simple adaptation template he/she must self keep track of time and use `GciTimeout` to mark a timer as timed out. If the user uses the timers adaptation template he/she can use the `AdSetTime` function (see below) to have this work done automatically.

GciSnapshotStatus GciPollingSnapshot()

This function is similar to the `GciSnapshot` function.

The main difference is that `GciPollingSnapshot` should poll the PCOs and Timers in the system, check the termination of the different test components, but should not stop and wait for any event above. If no events are done, `GciPollingSnapshot` should return, as opposed to `GciSnapshot`, that can be designed to stop and wait for an event to occur.

There is also a difference in the return value, `GciPollingSnapshot` will show if any event has occured or not, compared to `GciSnapshot`, where the return means that any event occured.

Return Value: `GciSnapshotStatus`

- `GciSnapshotEvent`, if everything in snapshot went fine and any event occured.
- `GciSnapshotNoEvent`, if everything in snapshot went fine and no event occured.
- `GciSnapshotNotOk`, if something catastrophic occured while doing the snapshot.

For backward compatibility with previous versions of adaptation, this function does not have to be implemented. For use with previous versions a dummy implementation exists in the TTCN static library which prevent linker errors.

`GciStatus GciSend(int pcod, GciValue* value)`

The user is responsible for implementing the send functionality. This involves identifying a given PCO and encoding the value before sending the message on that PCO.

`GciStatus GciStartTimer(int timerd, long duration, int unit)`

To start a timer the TTCN runtime behavior will call this function. Note that timer duration value is optional in TTCN but is always present here.

`GciStatus GciCancelTimer(int timerd)`

This function is called to stop the given timer. It should have the effect that the appropriate timer representation in the adaptation is stopped.

`long GciReadTimer(int timerd)`

This function is called when a timer is read. It must return the current timer value.

`GciStatus GciCreate(int ptc, char* tree, GciValue* args)`

This functions is called when a PTC is to be created to run a given dynamic behavior with the given arguments (concurrent TTCN).

`GciValue* <TS_Op>( <Arguments> )`

Every test suite operation must have this function defined. The name shall be the same as in the ATS and arguments must match the parameters in the ATS.

`GciStatus GciImplicitSend( GciValue* value )`

This function is called by the run-time behavior when an Implicit Send Event occurs in the test suite. It is up to the adaptation writer to define its exact implementation.

```
GciStatus GciLog(int logId, char* LogString)
```

The user has to implement parts of the logging facility. As the GCI document describes, all log identifiers are already implemented. User may change the logging for any event like it is desired. The log identifiers (identifies the type of log message) in this function should NOT be changed as they follow the values listed in the GCI document. The TTCN Runtime Behavior uses this function with these identifiers for logging.

logId	Description	logString must contain:
`Msg`	General message	Any information
`StartTC`	Start test case	The name of the test case
`StopTC`	Stop test case	The name of the test case
`Verdict`	Final verdict set	The verdict set
`PVerdict`	Prel. verdict set	The preliminary verdict set
`Match`	Line tried matches	Line number
`NoMatch`	Line tried does not match	Line number
`SendE`	Send Event	Line number The PCO name The type of message sent Constraint name and sent value
`RecE`	Receive Event[a]	Line number The PCO name The type of message received Constraint name and received value
`OtherE`	Otherwise Event	Line number The PCO name The type of message received Received value

logId	Description	logString must contain:
TimeoutE	Timeout Event	Line number The timer name
Assign	Assignment	Line number The left hand side The right hand side (textually)
StartT	Start timer	Line number The timer name The timer duration
CancelT	Cancel timer	Line number The timer name
ReadT	Read timer	Line number The timer name The timer value
Attach	Attachment: enter or exit test step	Line number Name of the attached Test Step Event type: Enter Test Step/Exit Test Step Test Step actual parameters
ImplSend	Implicit send	The sent value
Timeout	Timeout[b]	The timer descriptor number
Error	The error message	Error message text
Create	CREATE construct has been executed	Line number Component name Test Step name Test Step actual parameters
Done	DONE Event	Line number Component name(s)
Activate	Activate Event	Line number List of Defaults

logId	Description	logString must contain:
`MatchFailed`	A received line does not match.	Expected constraint name When logging is verbose, also constraint value and received value
`BeginMatchType`	When logging is verbose: Check that types and names have the constraint and incoming value or their corresponded parts, before trying to match them.	Constraint and incoming value: name, type name, base type name, class and tag.
`SuccMatchType`	When logging is verbose: The constraint and incoming value or their corresponded parts have been matched successfully.	Constraint name and type.
`FailMatchType`	When logging is verbose: The constraint and incoming value or their corresponded parts have not been matched.	Constraint name and type.
`SuccMatchValue`	When logging is verbose: The constraint and incoming value, or their corresponding parts, that were looked upon as simple values (the match process has not looked inside), have been matched successfully.	Constraint and incoming value

logId	Description	logString must contain:
GciLogFailMatchValue	When logging is verbose: The constraint and incoming value, or their corresponding parts, that were looked upon as simple values (the match process has not looked inside), have not been matched	Constraint and incoming value
FailAssignSignal	Generated by the SDL and TTCN Integrated Simulator when impossible to assign the incoming signal to the GCI value. The type on SDL and TTCN side thus has different descriptions	Part of incoming value that was assigned successfully to GCI value.
StartPTC	PTC has started	Test Step name
StopPTC	PTC has finished	Test Step name
CallQueue	When log option “Show executable path” is on, show Test Steps stack when new Test Step is started	Test Steps stack

a. Logging that a receive line matches.

b. Note this is low level logging done when the Test Adaptation writer calls GciTimeout

Value Interface

The interface to values is a simple API in which the user is allowed to build and access GciValues in an ordered way. The actual values used are not specified, only the methods that can be used on them are included. This way will allow vendors to have their own value representation within their TTCN Runtime Behavior and still conform to GCI.

Base Types/Values

These functions are used to transform actual values into the GCI representation. The interface is on base types only, so TTCN simple types are not visible in the interface.

```
GciValue* GciMkINTEGER( long num )
long GciGetINTEGER( GciValue* value )
```

These functions are used for INTEGER values and any simple type values of the base type INTEGER. These functions may be used for INTEGER values in the range of -2^31..2^31-1. Compare these functions with the GciMkBigINTEGER and GciGetBigINTEGER which are more universal and can be used for any unlimited INTEGER value.

Note:

The TTCN Suite RTS supports INTEGER values in the range of:
-2^32-1..2^32-1

```
GciValue* GciMkBigINTEGER(const char *value)
char* GciGetBigINTEGER(GciValue* value, char *buffer)
```

These functions should be used instead of GciMkINTEGER and GciGetINTEGER when the value is unknown or larger than what is allowed for the type long. The buffer should be large enough to store the given value (see the function GciGetBigINTEGERMaxSize).

The GciMkBigINTEGER function returns a new created value. The function will return NULL if there is an error.

A successful call of GciGetBigINTEGER will return a buffer. The function will return NULL if there is an error.

```
int GciGetBigINTEGERMaxSize(GciValue *value)
```

Returns the size of a buffer that is large enough for retrieving the character value of a value of the type INTEGER or ENUMERATED.

```
Bool GciIsBigINTEGERFitsLong(GciValue *value)
```

Returns GcTRUE when a value of the type INTEGER or ENUMERATED can be retrieved in a value of the type long (using the GciGetINTEGER function). The function will otherwise return GcFALSE.

```
GciValue* GciMkBOOLEAN( int bool )
int GciGetBOOLEAN( GciValue* value)
```

Used for Boolean values and for any simple type values with the base type Boolean.

```
GciValue* GciMkREAL(int mantissa, int base, int exponent)
GciReal GciGetREAL( GciValue* value)
```

Used for values of type Real and for any simple type values with the base type Real.

```
GciValue* GciMkBIT_STRING( const char* str )
char* GciGetBIT_STRING( GciValue* value)
```

Used for `bit string` values and for any simple type values with the base type `bit string`.

```
GciValue* GciMkHEXSTRING( const char* str )
char* GciGetHEXSTRING( GciValue* value)
```

Used for hex string values and for any simple type values with the base type hex string.

```
GciValue* GciMkOCTET_STRING( const char* str )
char* GciGetOCTET_STRING( GciValue* value)
```

Used for octet string values and for any simple type values with the base type octet string.

```
GciValue* GciMkNumericString( const char* str )
char* GciGetNumericString( GciValue* value)
```

Used for values numerical strings and for any simple type values with the base type numerical string.

```
GciValue* GciMkPrintableString( const char* str )
char* GciGetPrintableString( GciValue* value)
```

Used for Printable string values and for any simple type values with the base type Printable string.

```
GciValue* GciMkTeletexString( const char* str )
char* GciGetTeletexString( GciValue* value)
```

Used for Teletex string values and for any simple type values with the base type Teletex string.

```
GciValue* GciMkVideotexString( const char* str )
char* GciGetVideotexString( GciValue* value)
```

Used for Videotex string values and for any simple type values with the base type Videotex string.

```
GciValue* GciMkVisibleString( const char* str )
char* GciGetVisibleString( GciValue* value)
```

Used for Visible string values and for any simple type values with the base type Visible string.

```
GciValue* GciMkIA5String( const char* str )
char* GciGetIA5String( GciValue* value)
```

Used for IA5string values and for any simple type values with the base type IA5string.

```
GciValue* GciMkT61String( const char* str )
char* GciGetT61String( GciValue* value)
```

Used for T61string values and for any simple type values with the base type T61string.

```
GciValue* GciMkISO646String( const char* str )
char* GciGetISO646String( GciValue* value)
```

Used for ISO646string values and for any simple type values with the base type ISO646string.

```
GciValue* GciMkGraphicalString( const char* str )
char* GciGetGraphicalString( GciValue* value)
```

Used for Graphical string values and for any simple type values with the base type Graphical string.

```
GciValue* GciMkGeneralString( const char* str )
char* GciGetGeneralString( GciValue* value)
```

Used for General string values and for any simple type values with the base type General string.

```
GciValue* GciMkENUMERATED( int value )
int GciGetENUMERATED( GciValue* value)
```

Used for Enumerated values and for any simple type values of the base type Enumerated. These functions may be used with integer values of ENUMERATED values that are in the range of:

```
-2^31..2^31-1
```

The functions GciMkBigENUMERATED and GciGetBigENUMERATED are more universal and may be used for any unlimited ENUMERATED values.

Note:

The TTCN Suite RTS supports ENUMERATED values in the range of:
-2^32-1..2^32-1

```
GciValue* GciMkBigENUMERATED(const char *enumeration)
char* GciGetBigENUMERATED(GciValue* enumerated_object,
char *buffer)
```

These functions should be used instead of GciMkENUMERATED and GciGetENUMERATED when the value is unknown or larger than what is allowed for the type int. The buffer should be large enough to store the value (see the function GciGetBigINTEGERMaxSize).

The function GciMkBigENUMERATED returns a new created value. The function will return NULL if there is an error.

The function GciGetBigENUMERATED function returns a buffer when successful. The function will return NULL if there is an error.

```
GciValue* GciMkCHOICE(const char* name, GciValue* value)
GciValue* GciGetCHOICE( GciValue* value)
char* GciGetCHOICEName( GciValue* value)
```

Used for Choice values and for any simple type values with base type Choice.

```
GciValue* GciMkOBJECT_IDENTIFIER()
int GciOBJECT_IDENTIFIERSize( GciValue* value)
GciValue* GciAddOBJECT_IDENTIFIERComponent(
   GciValue* value, int comp)
int GciGetOBJECT_IDENTIFIERComponent( GciValue* value,
   int index)
```

Used for Object identifier values and for any simple type values with base type Object identifier.

```
GciValue* GciMkObjectDescriptor( const char* str )
char* GciGetObjectDescriptor( GciValue* value)
```

Used for ObjectDescriptor values and for any simple type values with base type ObjectDescriptor.

```
GciValue* GciMkNULL( )
int GciGetNULL( GciValue* value)
```

Used for Null type values and for any simple type values with base type Null type.

```
GciValue* GciMkANY( GciValue* value )
GciValue* GciGetANY( GciValue* value)
```

Used for ANY values and for any simple type values with base type ANY.

```
GciValue* GciMkR_TYPE( int value )
int GciGetR_TYPE( GciValue* value)
```

Used for R_Type values and for any simple type values with base type R_Type.

```
GciValue* GciMkPDU( GciValue* value )
GciValue* GciGetPDU( GciValue* value)
```

Used for PDU values and for any simple type values with base type PDU.

Base Functions

```
int GciGetType( GciValue* val )
```

Shall return the type of a value. The type is represented as the number used to identify global objects, see "Identifying Global Objects in TTCN/ASN.1" on page 1495. The function is valid for all values, but some values may be untyped in which case the function returns -1. These integer values used to identify all predefined and user-defined types are listed in the "type_gen.h" file generated by the TTCN C Code Generator. There is a function to get the symbolic type name from the integer value ("const char* GciGetTypeNameString(int TypeID)" on page 1519). A reverse function exists to get the integer type value from the symbolic name ("int GciGetTypeDescriptor(const char *type_name)" on page 1519).

```
GciValue* GciSetType( int type, GciValue* val )
```

Shall set the type of a value. The type is represented as the number used to identify global objects, see "Identifying Global Objects in TTCN/ASN.1" on page 1495. Valid for all values. Returns the input value with type set.

int GciGetBaseType (GciValue* val)

Shall return the base type (always one of the predefined types) of a value. The type is represented as the number used to identify global objects, see "Identifying Global Objects in TTCN/ASN.1" on page 1495.

const char* GciGetTypeNameString(int TypeID)

Shall return the symbolic name of a type with integer value `TypeID`.

int GciGetTypeDescriptor(const char *type_name)

Shall return the integer value for the type with the symbolic name `type_name`.

Value Management

The functions are divided into two meta sets derived from ASN.1: The `sequence` and the `sequence of`, corresponding to `struct` and `array` in C. The choice type of ASN.1 is not represented as a meta class because all values in the GCI value representation can be typed and therefore is the choice value implicit in GCI.

The functions only works within their intended set (e.g. GciSetField(GciMkSEQUENCE(2), 1, GciMkINTEGER()) is well defined but GciSetField(GciMkSEQUENCEOF(), 0, GciMkINTEGER()) is not defined and certainly is unpredictable.

Sequence and Set Types/Values

GciValue* GciMkSEQUENCE(int size)

Creates a sequence value with size children.

GciValue* GciSetField(GciValue* seq, int index, GciValue* fld)

Sets the field identified by index to the given value. Indices start at 1. The function is undefined for indices greater than the size of the sequence.

GciValue* GciGetField(GciValue* seq, int index)

Returns the field identified by index. Indices start at 1. The function is undefined for indices greater than the size of the sequence.

`int GciSeqSize( GciValue* seq )`

Returns the number of fields declared in the sequence *seq*.

Sequence of and Set of Types/Values

`GciValue* GciMkSEQUENCEOF()`

Creates a sequence of value.

`GciValue* GciAddElem( GciValue* seqOf, GciValue* elem )`

Adds the element *elem* to the end of sequence of *seqOf*.

`GciValue* GciGetElem( GciValue* seqOf, int index )`

Returns the element number *index* of the sequence of *seqOf*. Indices start at 0. The function is undefined for indices greater than current size.

`int GciSeqOfSize( GciValue* seqOf )`

Returns the current number of elements in *seqOf*.

Examples

This section contains examples of how the mapping between TTCN/ASN.1 and their GCI representation could be done. The examples use a conceptual model for encoding/decoding, no error handling is done and no attention is paid to the fact that some functions would use pointers because of allocation issues.

Global objects (PCOs and types) are identified with an unique number. This number is given a symbolic reference which is its TTCN name with a *D* for Descriptor appended to the end of it. The number for the PCO *L* is therefore referenced as *LD*.

Encoding/Decoding Examples

Each example consists of a table and the corresponding encode (and/or decode) functions. The examples focus on the value representation so the transfer syntax is simple: Each value is preceded by a header. The header contains the type of the value (as a number) and in some cases its length (element count, not size in bytes), in real ASN.1, the header would be built using tags. The header is written/read using primitives

BufWriteType, BufWriteSeqOfSize, etc. They are not encode/decode primitives but rather buffer primitives.

An encode function is a function that translates the GCI value onto a buffer. A decode function is one that reads a value from a buffer and builds a value in the GCI representation. Encoding functions are called from SEND and decoding functions are called from SNAPSHOT. The buffer has type *Buffer* and could be anything behaving as a sequence of bytes. The primitives BufReadInt, BufReadBool are used to read and write GCI basic values.

ASN.1 SEQUENCE Type

ASN.1 PDU Type Definition
PDU Name: T1
Comment: Example PDU for GCI examples
Type Definition
`SEQUENCE {` `  a INTEGER,` `  b BOOLEAN` `}`

Figure 273: Example

Encode

```
void EncodeT1( Buffer buf, GciValue* v)
{
  BufWriteType( buf, GcT1D );
  BufWriteInt( buf, GciGetField( v, 1 ));
  BufWriteBool( buf, GciGetField( v, 2 ));
}
```

Decode

```
void DecodeT1( Buffer buf, GciValue** v)
{
  int i;

  if ( BufReadType( buf ) != GcT1D )
    Error();
  *v = GciMkSEQUENCE( 2 );
  GciSetType( GcT1D, *v );
```

```
  i = BufReadInt( buf );
  GciSetField( *v, 1 , GciMkINTEGER( i ) );

  i = BufReadBool( buf );
  GciSetField( *v, 2 , GciMkBOOLEAN( i ) );
}
```

ASN.1 SEQUENCE OF Type

ASN.1 PDU Type Definition
PDU Name: T2
Comment: Example PDU for GCI examples
Type Definition
`SEQUENCE OF T1`

Figure 274: Example

Encode

```
void EncodeT2( Buffer buf, GciValue* v )
{
  int i;

  BufWriteType( buf, GcT2D );
  BufWriteSeqOfSize( buf, GciSeqOfSize( v ) );
  for (i = 1 ; i <= GciSeqOfSize( v ) ; i++ )
    {
      EncodeT1( buf, GciGetElem( v, i ) );
    }
```

Decode

```
void DecodeT2( Buffer buf, GciValue** v )
{
  int i, seqOfSize;
  GciValue* elem;

  if ( BufReadType( buf ) != GcT2D )
    Error();

  *v = GciMkSEQUENCEOF();
  GciSetType( GcT2D, *v );

  seqOfSize = BufReadSeqOfSize( buf );
```

```
  for ( i = 1 ; i <= seqOfSize ; i++ )
    {
      DecodeT1( buf, &elem );
      GciAddElem( *v, elem );
    }
}
```

ASN.1 CHOICE

ASN.1 PDU Type Definition
PDU Name: T3
Comment: Example PDU for GCI examples
Type Definition
`CHOICE {` `  c1 T1,` `  c2 T2` `}`

Figure 275: Example

Encode

```
void EncodeT3( Buffer buf, GciValue* v )
{
  const char *name = GciGetCHOICEName(v);
  GciValue *member = GciGetCHOICE(v);

  BufWriteType( buf, GcT3D );
  BufWriteString( buf, name );

  if(strcmp(name, "c1") == 0)
      EncodeT1( buf, v );
  else if(strcmp(name, "c2") == 0)
      EncodeT2( buf, v );
  else
      Error();
}
```

Decode

```
void DecodeT3( Buffer buf, GciValue** v )
{
  const char *name;
  GciValue *member;

  if ( BufReadType( buf ) != GcT3D )
    Error();
```

```
  name = BufReadString( buf );

  if(strcmp(name, "c1") == 0)
      DecodeT1( buf, &member );
  else if(strcmp(name, "c2") == 0)
      DecodeT2( buf, &member );
  else
      Error();

  *v = GciMkCHOICE((char *)name, member);
  GciSetType( GcT3D, *v );
  BufFreeString(name);
}
```

In-Line ASN.1 Type

ASN.1 PDU Type Definition
PDU Name: T4
Comment: Example PDU for GCI examples
Type Definition
SEQUENCE {   c1 T1,   c2 SEQUENCE {   c1 INTEGER,   c2 T2   } }

Figure 276: Example

Encode

```
void EncodeT4( Buffer buf, GciValue* v )
{
  GciValue* tmp;

  BufWriteType( buf, GcT4D );
  /* Encode first field */
  EncodeT1( buf, GciGetField( v, 1 ));
  /* Now encode inline type definition */
  tmp = GciGetField( v, 2 );
  BufWriteInt( buf, GciGetField( tmp, 1 ));
  EncodeT2( buf, GciGetField( tmp, 2 ));
}
```

Decode

```
void DecodeT4( Buffer buf, GciValue** v)
{
  int i;
```

```
  GciValue* tmp;
  GciValue* tmpseq;

  if ( BufReadType( buf ) != GcT4D )
    Error();

  *v = GciMkSEQUENCE( 2 );
  GciSetType( GcT4D, *v );

  DecodeT1( buf, &tmp );
  GciSetField( *v, 1, tmp );
  tmpseq = GciMkSEQUENCE( 2 );
  i = BufReadInt( buf );
  GciSetField( tmpseq, 1, GciMkINTEGER( i ) );
  DecodeT2( buf, &tmp );
  GciSetField( tmpseq, 2, tmp );
  GciSetField( *v, 2, tmpseq );
}
```

TTCN Examples

`myQueue` below is the Test Adaptation writers own representation of the PCO queue(s). In this example there is only one PCO, called `L` (`Ld` for its number). Note that the number `Ld` can be any number (most certainly not zero).

Snapshot Example

Snapshot must read the status of the IUT and tell this to the TTCN Runtime Behavior.

```
void GciSnapshot()
{
  GciValue* v;

  /* Check if anything has arrived,  */
  /* This would be a while statement */
  /* in a blocking context           */
  if ( ! myQueue[ 0 ].empty )
    {
      switch ( BufPeekType( myQueue[0].buf ))
        {
        case GcT1D:
          DecodeT1( myQueue[0].buf, &v );
          break;
        case GcT2D:
          DecodeT2( myQueue[0].buf, &v );
          break;
        default:
          Error();
        }
```

```
      GciReceive( GcLD, v );
    }

  /* Nothing has happened. */
}
```

Send Example

For the send example the following ASP table is used to show an API view of encode.

ASN.1 ASP Type Definition
ASP Name: TCONreq
Comment: Example ASP for GCI examples
Type Definition
SEQUENCE {   num INTEGER, -- sequence number   pdu T1 -- Embedded pdu }

Figure 277: Example

```
/* Two examples of send functionality */
GciStatus GciSend( int pcoDescr, GciValue* msg )
{
  if ( pcoDescr == GcPcoToSocketD )
    {
      /* Encode first */
      if ( GciGetType( msg ) == GcT1D )
        {
          EncodeT1( buf, msg );
          BufSocketSend( buf );
        }
    }
  else if ( pcoDescr == GcPcoToAPID )
    {
      if ( GciGetType( msg ) == GcTCONreqD )
        {
          Buffer pdu;
          EncodeT1( pdu, GciGetField( msg, 2 ));
          /* Call to lower layer */
          T_CONreq( GciGetField( msg, 1 ), pdu );
        }
    }
```

EGci Value Construction and Functions

The GCI interface has been defined as a base for executable test suite adaptation. One part of this interface covers values, how to build them, access them, etc. While this interface was not enough, a set of functions has been added with the EGci prefix so that the GCI functions (a set defined by a standard) was not changed. The purpose of this section is to describe this paradigm of value construction and the current EGci functions available.

Value Construction

The major difference in value handling is how values are constructed. Values are either created by the GciMk-prefixed functions, followed by a GciSetType call, or value are created by a call to the EGciMkValue function. The run time system is compatible with both alternatives but the latter is preferred. The difference is that the GciMk-function creates untyped values of a given predefined type (SEQUENCE, SET, INTEGER, etc.) while the EGciMkValue takes the type descriptor of the user-defined type and creates the complete value structure with correct types throughout the whole value structure. Values of type SEQUENCE OF, SET OF and CHOICE cannot be fully instatiated since the size/choice of the value is not known at code generation time. Elements of the SEQUENCE/SET type are appended explicitly and CHOICE values are also selected explicitly.

The procedure is to create the value structure (with uninitialized leaf values) and then extract the proper value (container) through ordinary field access and extraction functions. Then the value is assigned using the EGciAssign function. When available, for simple non-structured values, EGciSet/GciSet-functions can also be used on the extracted values.

Below you will find two alternatives for creating a value given the following type:

```
T ::= SEQUENCE
{
  a SEQUENCE {
     b MyINTEGER
  },
  c MyINTEGER
}

MyINTEGER ::= INTEGER
```

```
v T := { a { b 17 }, c 42 }
```

Alternative 1 – Using GciMkSEQUENCE and GciSetType

The individual values through out the structured value have to be created and the correct type set. It is also impossible to set the correct type for the inlined types since they are unknown.

```
...
value = GciMkSEQUENCE(1);
GciSetType(value, GcTD);

inner_seq = GciMkSEQUENCE(1);
/* GciSetType(inner_seq, ?); This is not possible
since the inlined type is not available to the user.
A generated no-name type exists but is not known. So
the SEQUENCE will be untyped (only a SEQUENCE). This
is not a problem with alternative 2. */

b = GciMkINTEGER(17);
GciSetType(b, GcMyINTEGERD);
GciSetField(inner_sequence, 1, b);
c = GciMkINTEGER(42);
GciSetType(c, GcMyINTEGERD);
GciSetField(value, 1, inner_seq);
GciSetField(value, 2, c);
...
```

Alternative 2 – Using EGciMkValue (Recommended Approach)

Using the EGciMkValue function makes it somewhat more readable, but what is more important is that the final value has the correct type throughout the structure (all done by the generated type information used in the call to EGciMkValue). All field names are also correct.

```
...
value = EGciMkValue(GcTD);
/* With this call the whole structure is created and
the basic leaf values are only extracted and set
("filled in"). */

tmp = GciGetField(value, 1);
tmp = GciGetField(tmp, 1);
EGciSetINTEGER(tmp, 17);

tmp = GciGetField(value, 2);
EGciSetINTEGER(tmp, 42);
...
```

Available Functions

GciValue * EGciMkValue(int)

Takes the generated type identifier constant `Gc<type>D` and creates a proper value using all available type information generated. This function is faster than the `EGciMkValueFromTypeName` but the type identifier constants can potentially change their numeric value if new types are added to the test suite. Any depending encoder/decoders have to be recompiled if the generated type information changes. If generation dependent information is required, the `EGciMkValueFromTypeName` should be used.

Example 292 ----------

Given the type `MyType`, the call for constructing a value would be `EGciMkValue(GcMyTypeD)`.

GciValue * EGciMkValueFromTypeName(const char* tname)

Same as `EGciMkValue` but it takes the (bare) name of the type.

Example 293 ----------

Given the type `MyType` the call for constructing a value would be `EGciMkValueFromTypeName("MyType")`.

----

void EGciRmValue(GciValue *value)

This function deletes a constructed value.

GciStatus EGciAssign(GciValue *destination, GciValue *source)

Given the two values, which must be instantiated values of the same type, the source is assigned to the destination value.

GciValue * EGciGetFieldByName(GciValue *val, const char* name)

This function returns the named member of a SEQUENCE, SET or CHOICE value. If the value given by the parameter `val` has no member with the name given by the parameter `name`, the return val-

ue is NULL. Note that the return value is not a copy of the value member, so if any operations are applied to the return value, this will directly effect the member value.

GciStatus EGciSetMemberByName(GciValue *val, const char* name, GciValue *mem_val)

Sets the named member of a SEQUENCE, SET or CHOICE value to the given value (mem_value). This is the same as doing an extraction followed by an assignment.

GciStatus EGciSetEmptySET_OF(GciValue *value)

Sets the value to the defined state. That is, it is an empty SET OF value.

GciStatus EGciSetEmptySEQUENCE_OF(GciValue *value)

Sets the value to the defined state. That is, it is an empty `SEQUENCE OF` value.

GciStatus EGciSetOMIT(GciValue *value)

Sets the given optional value as omitted.

GciStatus EGciSetBOOLEAN(GciValue *value, Bool v)

Sets a constructed value of the (base) type `BOOLEAN` to the given boolean value.

GciStatus EGciSetSTRING(GciValue *value, const char *)

Sets a constructed value of any string base type (`BITSTRING`, `OCTETSTRING`, `IA5String`, etc.) to the given string value.

GciStatus EGciSetINTEGER(GciValue *value, int number)

This function sets a constructed value of the (base) type `INTEGER` to the value given by the `number` parameter. This function may be used for `INTEGER` values in the range of `-2^31..2^31-1`. The function `EGciSetBigINTEGER` is more universal and is used for unlimited `INTEGER` values.

Note:

In the TTCN Suite RTS there is support for `INTEGER` values in the range of:
`-2^32-1..2^32-1`

GciStatus EGciSetBigINTEGER(GciValue *value, const char *number)

This function sets a constructed value of the (base) type `INTEGER` to the given integer value.

This function should be used instead of `EGciSetINTEGER` when the value is unknown or larger than what is allowed for the type `int`.

GciStatus EGciSetNULL(GciValue *value)

This function sets a constructed value of the (base) type `NULL` to a `NULL` value.

GciValue * EGciSetENUMERATEDByName(GciValue *val, const char* enum_name)

This function sets the actual value of the ENUMERATED value to one of its declared named numbers. For example, EGciSetENUMERATEDByName(gcivalue, “foo”), where “foo” is one of the named numbers in the type.

Example 294 ----------

```
EType ::= ENUMERATED { first, second, third }

GciValue *eval = EGciMkValue(GcETypeD);
EGciSetENUMERATEDByName(eval, "second");
```

----

const char * EGciGetENUMERATEDName(GciValue *value)

Retrieves the name of the current value.

Example 295 ----------

Continuing Example 294, a call to `EGciGetENUMERATEDName(eval)` would return the string `"second"`.

GciValue * EGciGetChoiceMemberByName(GciValue *value, const char *name)

CHOICE values are not constructed all the way down to its leaf value since the information about the selected field is not available in the type information. When this function is called, the value construction is continued and proper value will be constructed for the

selected CHOICE field (name) and the value can then be assigned properly.

GciValue * EGciGetChoiceMemberByTag(GciValue *value, GciTagClass tag_class, int tag)

Equivalent to `EGciGetChoiceMemberByName` but the unique tag of the choice element instead of the name is used.

GciValue * EGciGetSEQUENCE_OFElement(GciValue *value)

Constructs a new value given the underlying/contained type of the SEQUENCE_OF type. The newly constructed value must be appended using the `GciAddElem` function.

GciValue * EGciGetSET_OFElement(GciValue *value)

Constructs a new value given the underlying/contained type of the SET_OF type. The newly constructed value must be appended using the `GciAddElem` function.

Bool EGciGetAnyValue(GciValue *value)

Predicate to check if the value is ANY value attribute. Note that this has nothing to do with the `GciGetANY` function since that function operates on values of the type `ANY`.

Bool EGciGetAnyOrOmit(GciValue *value)

Predicate to check if the value is ANYOROMIT value attribute.

Bool EGciGetIfPresent(GciValue *value)

Predicate to check if the IF_PRESENT value attribute is specified for this value.

Bool EGciGetDefault(GciValue *value)

Predicate to check if the instatiated value comes from the default value of the type or not.

Error Handling

If an error occurs during execution, the error state is propagated up through the call stack and error information is added. An error has occurred when either the EGciGetErrorCount function returns a number greater than zero or a Gci/EGci-function call has returned GciNotOk.

The error message is retrieved by EGciGetLastErrorMessage and reset by EGciClearError.

unsigned int EGciGetErrorCount()

Retrieve the current number of errors detected by the run time system.

const char* EGciGetLastErrorMessage(void)

Retrieve the last error message in text form.

void EGciClearError()

Clear the error state of the run time system.

Miscellaneous

void EGciDumpValue(FILE *stream, const char *prefix, GciValue *value, const char *suffix)

Prints the given value on stream. The prefix and suffix are printed before and after the value dump respectively.

int EGciDumpValueInBuffer(char* buffer, int maxlen, GciValue* value);

Returns 0 when `value` was successfully dumped into `buffer`. If the function fails it returns the required buffer size for a successful dump of `value`.

This function can be used for printing a value from the code, for example a value in a GciValue structure to be printed in a file. Call `EGciDumpValueInBuffer` with a large enough buffer (possibly first check the necessary buffer size), then print the value from `buffer`.

void EGciSetDebugStream(FILE *stream)

Set the debug stream where logging will be made. It is set to `stderr` by default.

Examples

```
MyString         ::= IA5String
EmailAddress     ::= MyString
SnailmailAddress ::= SET { street MyString, number INTEGER }
AddressKind      ::= ENUMERATED { email, snailmail }

-- This represents a contact with name and an address.

Contact ::= SEQUENCE {
  name MyString,
  address_kind AddressKind
  address CHOICE {
    email     EmailAddress,
    snailmail SnailmailAddress
  }
}
```

A value would be constructed in the following way:

```
GciValue *tmp;
GciValue *address;
CciValue *contact;

contact = EGciMkValue( GcContactD);

tmp = EGciGetFieldByName(contact, "name");
EGciSetSTRING(tmp, "Elvis");

tmp = EGciGetFieldByName(contact, "address_kind");
EGciSetENUMERATEDByName(tmp, "snailmail");

 address = EGciGetFieldByName(contact, "address");
 address = EGciGetChoiceMemberByName(address, "snailmail");
 tmp = EGciGetFieldByName(address, "street");
 EGciSetSTRING(tmp, "High road");
 tmp = EGciGetFieldByName(address, "number");
 EGciSetINTEGER(tmp, 42);
```

The Adaptation Framework

Introduction to the Adaptation Framework

The Adaptation Framework, technically referred to as “ACM”, is a collection of the functions that are needed to write a full-fledged adaptation implemented by plug-in libraries that can be used “as is”. Currently, the following plug-in modules are included in the TTCN Suite distribution:

- TCP/IP socket communication implementation.
- Standard system-time timer implementation.

This means that if the target system uses TCP/IP to communicate, the adaptation writer can simply use the Adaptation Framework right from the box to get the IUT connected to generated code with minimal amount of work.

Note that the Adaptation Framework is provided as a standardized way to implement the GCI interface. This means that the Adaptation Framework *does not replace GCI, but rather complements it.* To build the adaptation, simply implement the needed GCI functions by calling the counterpart framework functions that use the compiled-in implementations for the low level code; in this distribution this would be the TCP/IP protocol for communication and a standard system time timer implementation. On the other hand, this means that one can continue to implement adaptations without using the framework altogether, the old way, and old adaptations will continue to work as before.

Examples of usage

The included adaptation residing under the ACM/ subdirectory contains an implementation of the GCI layer by framework functions and is instructive in displaying the way to work with framework functions.

Function reference

Communication data types

Defined constant values

Defined constant	Meaning
ACM_ADDRESS_TCPIP	The address type specifier used with the TCP/IP implementation of the Adaptation Framework for the `GciAddress` type.

More values will be defined in the file *acm.h* once more plug-in packages are available. The user can also specify new values for his/her own communication modules.

> **Note:**
> Values 0-100 are reserved for use by IBM Rational.

Communication role of the executable test suite

```
typedef enum
{
  ACMClient,
  ACMServer
} ACMConnectionType;
```

The enumerated type *ACMConnectionType* is used when calling `ACMConnect()` to select whether the generated code should act as a server or client to the IUT.

Communication primitives

These functions are used to implement the GCI communication functions.

Initializing the communication and timer package

```
GciStatus ACMInit(const GciTime* max_timeout,
GciTimeUnit time_tick_unit)
```

Must be called after GCIInit() but prior to any use of the ACM communication or timer primitives. The first argument, *max_timeout* defines

the maximum amount of time the system will wait in blocked state for any timer. Setting this to a low value enables the possibility to write a polling snapshot function. A defined constant exists by the name of **ACM_ONE_YEAR_IN_MINUTES** that can be used as a default “big enough” number argument.

The second argument, *time_tick_unit*, depicts the unit of time (in GCI time units) used internally in the run-time system. Since all internal timer data is converted to and from this unit, it is important to set this to a value that encompasses approximately the value range of the time-out values used in the test suite. This parameter also restricts the maximum time that is available for a time-out value, as depicted below.

Internal unit	Maximum length of time before timer wraps
Gcips	4.3 seconds
Gcins	71 minutes
Gcius	49 days
Gcims	136 years
...	...

Resetting the Run-time System

```
GciStatus ACMReset()
```

This function cancels all timers and resets all timer queues and PCO buffers. It is typically called before starting a new test case to ensure that all old data is removed and to put the run-time system into an initial mode.

Registering the GciTimeout Function

```
GciStatus ACMRegisterTimeoutHandler(ACMTimeoutHandler
                                    timeouthandler)
```

This function needs to be supplied with a pointer to the *GciTimeout()* function before using the *ACMSnapshot()* function, since *ACMSnapshot()* calls the timeout function automatically for every timer that has expired.

Registering the GciReceive Function

```
GciStatus ACMRegisterReceiveHandler(GciReceiveHandler
                                    receive_callback)
```

This function needs to be supplied with a pointer to the *GciReceive()* function before using the *ACMSnapshot()* function, since *ACMSnapshot()* calls the receive function automatically for every data packet that has arrived.

Registering the Active Decode Function

```
GciStatus
ACMRegisterDefaultDecodeHandler(ACMDecodeHandler
                                decode_handler)
```

The function *ACMSnapshot()* calls *GciReceive()* automatically for every data packet that are ready to be received. In order to do this, however, a decode function needs to be registered that can decode the incoming data from its transfer syntax to the internal GCI value representation. This is easily done by the above call. Note that this opens the possibility to switch decoding during runtime, since any decode function that fulfills the requirements are eligible to be registered as the default decoder at any time. See "Encoding, and in Particular Decoding within ACM and GCI" on page 1556 in chapter 36, *Adaptation of Generated Code* for details on how to construct the decode function(s).

Registering a Specific Decode Function for a PCO

```
GciStatus ACMRegisterDecodeHandler(ACMDecodeHandler
                                   decode_handler,
                                   GciPCOID pco_id)
```

This function is similar to the previous function, but it enables a certain decode function to be connected with a certain PCO. This means that the function *ACMSnapshot()* will be able to use different decoding mechanisms depending on which PCO the message was received from. See "Encoding, and in Particular Decoding within ACM and GCI" on page 1556 in chapter 36, *Adaptation of Generated Code* for details on how to construct the decode functions.

Connecting to a Communication Port

```
GciStatus ACMConnect(GciPCOID pco_id,
                     GciAddress* address,
                      ACMConnectionType type,
                      unsigned int buffer_size)
```

This function maps the PCO identifier value internally to an external communication port and makes the connection. This makes it possible to use the PCO identifier in all subsequent calls, which makes the code quite clear.

In addition, we need to supply what type of role the ETS will have in the communication with the IUT - as a server (ACMServer) or as a client (ACMClient).

The *buffer_size* argument defines the largest possible buffer (in bytes) that can be received in one chunk. This should usually be set to the largest buffer needed to encode a value in the current value representation.

Disconnecting a Communication Port

```
void ACMDisconnect(GciPCOID pco_id)
```

This function disconnects a previously opened communication link.

Sending a Message

```
GciStatus ACMSend(GciPCOID pco_id, GciBuffer* buffer)
```

This function sends an encoded buffer on the designated PCO. When sending, the function will block until the entire buffer have been sent, or an error has occurred. This is simply because it is preferred to see the send operation as an atomic operation without possible race conditions.

Receiving a Message

```
GciStatus ACMReceive(GciPcoID pco_id, GciBuffer* buffer)
```

This function tries to receive a message from the designated PCO. If successful, the received encoded byte stream is inserted to the provided buffer, which is allocated inside the function.

Retrieving the Position of the First PCO with Received Data

```
GciPosition ACMGetReceivedPCOPos()
```

This function returns the position of the first PCO to have data ready for receiving. The return value is simply meant to be used as an iterator argument to the `ACMGetNextReceivedPCO()` function. If the returned value is zero, there are at this time no PCO that has received data.

Retrieving the position of the next PCO with received data

```
GciPCOID ACMGetNextReceivedPCO(GciPosition* position)
```

This function returns the numeric identifier of the PCO that has data ready to receive and updates the *position* variable to point to the next PCO with data to receive. If the *position* is zero, no further PCO has data to receive.

A simple loop that checks all PCO's for receive can be written in a similar manner to the timer iteration described below in "A Loop That Fetches All Expired Timers and Prints Their ID Numbers" on page 1542 in chapter 36, *Adaptation of Generated Code*.

Timer Primitives

Timer handling can be implemented in various ways. The package makes it easy for the adaptation writer to use timers, by encapsulating functionality that used to be implemented in a fairly standard way in GciSnapshot() implementations over and over again. Instead of using cryptic timer structures the user can now use the GCI identifier for the needed timer in a number of timer access functions. Time itself is fetched from the system clock, normalized to zero at the time of starting the execution.

Starting a Timer

```
GciStatus ACMStartTimer(GciTimerID timer_id,
GciTime timeout)
```

This function starts the designated timer with the given expiration time. If the timer has not yet been created, this call will create it. If the timer was running or expired it is simply restarted.

Cancelling a Timer

```
GciStatus ACMCancelTimer(GciTimerID timer_id)
```

This function transfers the designated timer from the active list to the list of stopped timers. This call needs to be performed once a timer has been noted as expired (via `GciTimeout()` in `GciSnapshot()` for instance)

Cancelling All Timers

```
GciStatus ACMCancelAllTimers()
```

This function cancels all timers, even if they are already expired.

Reading the Value of a Timer

```
GciStatus ACMReadTimer(GciTimerID timer_id,
GciTime* timer_before_timeout)
```

This function places the current time of a given timer in the provided GciTime object.

Checking the Current Status of a Timer

```
GciStatus ACMTimerStatus(GciTimerID timer_id,
GciTimerStatus* timer_status)
```

This function places the current status of the given timer in the status variable. Please refer to the declaration of GciTimerStatus for the various states of a timer.

Retrieving the Key to the First Timer That Has Expired

```
GciPosition ACMGetTimedOutPos()
```

This function retrieves the position to the first timer that has expired from a chronological list of expired timers. The return value is simply meant to be used as an iterator argument to the `ACMGetNextTimed-Out()` function. If the returned value is zero, there are no currently expired timers.

Getting the Next Expired Timer

GciTimerID ACMGetNextTimedOut(GciPosition* position)

Given a position index (the index to the first timer is fetched with the ACMGetTimedOutPos() function), this function returns an expired timer and updates the position variable to point to the next timer in the list of expired timers. If the position value is equal to zero after this call, no more expired timers exist in the list.

Example 296 A Loop That Fetches All Expired Timers and Prints Their ID Numbers

```
. . .
GciPosition pos = ACMGetTimedOutPos()

while (pos != NULL) {
  printf("Timer %d has expired!\n",
         ACMGetNextTimedOut(&pos) );
}
```

Time Left Before the Next Timer Expires

GciStatus ACMTimeLeft(GciTime *time_before_timeout)

This function retrieves the actual time left before the next timer is due to expire. If one or more timers have already expired, the function returns zero as the time left. If no timers exist, or all timers have been stopped, the function returns GciNotOk.

Timer and PCO Handling with a Single Call

GciStatus ACMSnapshot()

For the hardened writer of many adaptations, this function offers a relief. Calling this function will wrap most GciSnapshot functionality into one call. Briefly, what the function performs internally is the following:

1. Check all PCO/CP channels for received data.
2. For each queue that contains fresh data, receive, decode it and call GciReceive() with the result.
3. For each expired timer, call GciTimeout() with the expired timer's ID number and move the timer to the list of stopped timers.
4. Return status of the operations.

In effect, the simplest snapshot implementation can now look like this:

Example 297 The Simplest Snapshot in the World!

```
GciStatus GciSnapshot()
{
  return ACMSnapshot();
}
```

Somewhere in the code, before calling ACMSnapshot() for the first time, it is also necessary to register three functions with ACM:

- GciReceive()
- GciTimeout()
- A decode function.

This is due to the fact that ACMSnapshot() will possibly try to call these functions depending on the internal state. A good place to register these functions is close to the ACMInit() call:

```
.
.
ACMInit(max_timeout, ... );
.
.
ACMRegisterTimeoutHandler(&GciTimeout);
ACMRegisterReceiveHandler(&GciReceive);
ACMRegisterDefaultDecodeHandler(&DecoderFunction);
```

Waiting for the Next Event

`GciStatus ACMWaitForEvent()`

If more functionality is needed in the snapshot function, a mechanism for waiting for the correct amount of time is needed. This function sleeps until the next timer is due, or until data arrives on a PCO, whichever comes first. The function then returns with the status GciOk, at which point the snapshot function can simply poll the input queues for data, and/or take the now arrived timeout. If no timers are running, the function returns after the maximum time to wait as defined in **ACMInit(...)**. If this time is set to a small amount, there is a polling snapshot in effect.

Example 298: The Same Snapshot but with Written Out Code ––––––

```
GciStatus GciSnapshot()
{
  GciPosition expired_pos;
  GciTimerID  expired_timer;
  GciPosition received_pos;
  GciPCOID    received_pco;
  GciBuffer   buffer;
  GciValue*   value;

/* Sleep until either a timer has expired
 * or data is received. For a polling
 * snapshot, remove this. */
  if (ACMWaitForEvent() != GciOk) {
    return GciNotOk;
  }

/* First, check all timers. */
  if (ACMSynchronizeTimers() != GciNotOk) {
    return GciNotOk;
  }
  expired_pos = ACMGetTimedOutPos();

  while(expired_pos != NULL) {
    expired_timer =
       ACMGetNextTimedOut(&expired_pos);
    GciTimeout(expired_timer);
    if (ACMCancelTimer(expired_timer) != GciOk) {
      return GciNotOk;
    }
  }
/* Next, take care of the PCO's */

  buffer.buffer =
   (char *)malloc(sizeof(char) *
                  MAX_ENCODING_BUFFER+1);
  buffer.current_length = 0;
  buffer.max_length = MAX_ENCODING_BUFFER;

  received_pos = ACMGetReceivedPCOPos();

  while(received_pos != NULL) {
    received_pco =
      ACMGetNextReceivedPCO(&received_pos);
    if (ACMReceive(received_pco, &buffer) != GciOk)
    {
      return GciNotOk;
    }
    value = GeneralDecode(&buffer);
    if (value != NULL) {
      if (GciReceive(received_pco, value)!= GciOk) {
        return GciNotOk;
      }
    }
```

```
      else {
        fprintf(stderr,
                "Transfer syntax error on PCO %s\n",
                GciGetPCOName(received_pco) );
        return GciNotOk;
      }
    }
    return GciOk;
  }
```

Retrieving the Current System Time

`GciStatus ACMCurrentTime(GciTime* current_time)`

This function retrieves the time of the system clock in the instance of the function call.

Error Handling

The error handling of the Adaptation Framework will be straightforward to those familiar with the *err_no* paradigm of C. Every time an error occurs, the function sets an internal error code and returns *GciNotOk*. The error code can then be retrieved either as an enumerated value, or as a descriptive string.

The error codes can be found in the file *acm.h*.

Setting the Error Code

`ACMSetLastError(ACM_Error_Number errornumber)`

Sets the error number to the given value.

Retrieving the Error Code of the Last ACM Error

`ACM_Error_Number ACMGetLastError()`

Returns the last error code as an enumerated value.

Retrieving the Error String of the Last ACM Error

`const char* ACMGetLastErrorMessage()`

Returns the last error code as a printable string.

Completing the Adaptation

The generated code has to be extended before it is complete in order to test the intended implementation. This section describes how to make these extensions.

Figure 278 displays in a simple way the anatomy of an executable test suite.

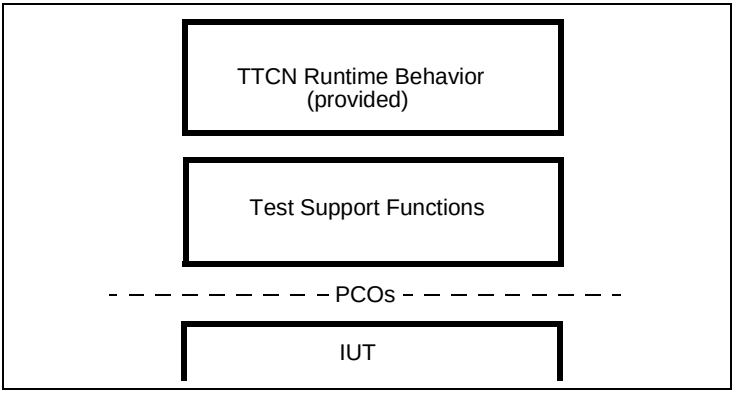


Figure 278: The anatomy of an executable test suite

When code is generated by the code generator, it does not know anything about the system it is about to test. It assumes that it will have access to certain functions, implemented by the user. This is the “Test Support Functions” module in Figure 278.

One important thing to remember is that the previously defined interface called GCI, is a standardized set of functions.See “The GCI Interface” on page 1488 in chapter 36, *Adaptation of Generated Code*. **Adaptation should be made using the functions and data types defined by that interface**.

The Test Support Functions

This is the “glue” between the TTCN Runtime Behavior and the IUT. It is a set of functionality (functions) that is adaptation specific and should be provided by the user.

The following areas have to be covered and they are described in more detail in the sections they refer to.

Implementation and Handling of Timers

The timers defined in the Test Suite must have a real representation in the test environment. The TTCN Runtime Behavior will, when necessary, ask the adaptation about the status of a timer. See "Representation and Handling of PCO and CP Queues" on page 1550 in chapter 36, *Adaptation of Generated Code*.

Representation and Handling of PCOs [and CPs]

Messages are communicated through *Points of Control and Observation* and *Connection Points*. The actual buffering of incoming IUT messages must be supplied by the adaptation. See "Representation and Handling of PCO and CP Queues" on page 1550 in chapter 36, *Adaptation of Generated Code*.

Communication with Test Equipment [and PTCs]

To be able to test the IUT at all, there must be actual communication channels to and from it. The actual communication is of course totally depending on the system environment. See "IUT Communication" on page 1549 in chapter 36, *Adaptation of Generated Code*.

Encoding and Decoding

Values in the test suite (constraints, variables, constants, etc.) must be properly encoded and decoded for sending them to the IUT. The actual protocol for encoding decoding is up to the user. See "Encoding and Decoding" on page 1551 in chapter 36, *Adaptation of Generated Code* and "Encoding and Decoding Using BER" on page 1558 in chapter 36, *Adaptation of Generated Code* for more details.

Implementing the Adaptor the Efficient Way – the Adaptation Framework

The implementation of timer handling and communication primitives can be made by calls to the Adaptation Framework API, which permits the underlying "hard" protocol and timer implementation to change without having to change the high-level adaptation code. See "The Adaptation Framework" on page 1535 in chapter 36, *Adaptation of Generated Code* for a thorough explanation.

Timers

Timers need some special attention. As timers are implemented differently on different systems the implementation of the timers might differ. See the *timers* adaptation template in the installation for ideas.

Timers are simply a number of constructions to keep track of each of the test suite timers. In the generated code a timer is represented by an integer descriptor which uniquely identifies it. The timer implementation supplied with the Adaptation Framework implements the timeout functionality by maintaining an ordered list of the running timers, which means that retrieving the time left to timeout and the next timer to expire are quick operations.

To refresh the current status of timers, the timer lists need to be *synchronized* to the system time. This is made automatically by the snapshot function `ACMWaitForEvent()` (more on this function later) or can be made explicitly by a call to the function `ACMSynchronizeTimers()`. This function simply checks all running timers with the system clock, moving timers that are due to the list of expired timers.

There are four functions to consider regarding timers:

```
GciStartTimer

GciCancelTimer

GciReadTimer

GciSnaphot
```

The interfaces to timers that can (and should) be called in the TTCN Runtime Behavior are:

```
GciTimeout

GciGetNoOfTimers

GetTimer

GciGetTimerName

GetTimerIndex
```

For details concerning these functions, see "GCI C Code Reference" on page 1503 in chapter 36, *Adaptation of Generated Code*.

The Adaptation Framework interfaces to timer functionality that can be used to implement the GCI functions are:

```
ACMStartTimer
ACMCancelTimer
ACMCurrentTime
ACMGetTimedOutPos
ACMGetNextTimedOut
ACMReadTimer
ACMTimeLeft
ACMTimerStatus
ACMWaitForEvent
```

These functions map to a real-time timer implementation that is provided with the installation of TTCN Suite and can be used “as-is”. See Release Notes for details on which platforms are supported by this timer package.

Timer Adaptation Example

The *timers* adaptation template is the simple adaptation template with timers implemented. It demonstrates in general how to use the Adaptation Framework to implement timer functionality, and in particular the use of the call `ACMSynchronizeTimers()` that must be used in snapshot when `ACMWaitForEvent()` cannot be used.

IUT Communication

From an abstract point of view, sending and receiving is done over PCOs (Points of Control and Observation). The physical representation of these PCOs has to be defined by the user. It can be shared memory, serial communication, sockets, etc. This, of course, is only done on the controlling side. The PCOs have to be connected somewhere to the test equipment and the responsibility for this is put upon the user.

The GCI functions that must perform the communication is (at least):

```
GciSend
```

```
GciReceive
```

For details concerning these functions, see "GCI C Code Reference" on page 1503 in chapter 36, *Adaptation of Generated Code*.

Representation and Handling of PCO and CP Queues

PCOs are constructions to handle the PCO queues. Each PCO should have buffers for sending and receiving, a method for retrieving the status of the receive buffer and additional information such as channels and ports must be provided for the physical channels.

If the ETS is to run within the test equipment, i.e. the communication between the ETS and the IUT resides within the test equipment, the ETS has to be moved (cross compiling or if possible compiling within the test equipment).

This queue initialization should of course be made before any test case are run. The function that concerns the PCO's and CPs are:

```
GciSnapshot
```

```
GciSend
```

`GciSnapshot()` can be implemented using for instance `ACMWaitForEvent()`, and `GciSend()` can be implemented using `ACMSend()`. Today, the ACM layer builds upon a TCP/IP socket implementation but other protocol implementations can be easily added.

The interface that should be used when a message has been received (after proper decoding) in the TTCN Runtime Behavior is:

```
GciReceive
```

Note that if the ACM layer function `ACMWaitForEvent()` is used in the snapshot function, the user never has to call `GciReceive()` since this is done automatically from within `ACMWaitForEvent()`!

For details concerning these functions, see "GCI C Code Reference" on page 1503 in chapter 36, *Adaptation of Generated Code*.

Encoding and Decoding

tBuffer is the subsystem of UCF that is used to represent values of ASN.1 types in the transmitted bit pattern. This section describes the interface for tBuffer.

Functions

To invoke functions of tBuffer, the following macros are to be called:

Initialization of buffer

You should initialize tBuffer before using it.

```
void BufInitBuf(tBuffer buf, tDirectionType dirtype,
UCFStatus status)
```

tBuffer buf	Buffer for data storage
tDirectionType dirtype	Data writing order (DIRECT or REVERSE).
UCFStatus status	Status of initialization. Variable “status” is equal to UCF_Ok if no errors were encountered during initialization or an error code otherwise.

Note: Buffer direction mode

`REVERSE` modes is not supported in the buffers anymore, although the prototype of the `BufInitBuf` function still includes the `dirtype` parameter for backwards compatibility. The direction mode parameter is always ignored when calling the `BufInitBuf` function.

Closing of buffer

You should close tBuffer after working with it.

```
void BufCloseBuf(tBuffer buf)
```

tBuffer buf	Buffer

Initialization in write mode

This function initializes tBuffer in write mode. You can write data to the buffer after calling this function.

```
UCFStatus BufInitWriteMode(tBuffer buf)
```

Initialization in read mode

This function initializes tBuffer in read data. You can read data from the buffer after calling this function.

```
UCFStatus BufInitReadMode(tBuffer buf)
```

Closing write mode

This function closes write mode of tBuffer. You should call this function after the 'write to buffer' operation is completed.

```
void BufCloseWriteMode(tBuffer buf)
```

Closing read mode

This function closes read mode of tBuffer. You should call this function after reading from the buffer is finished.

```
void BufCloseReadMode(tBuffer buf)
```

Getting direction type of the buffer

This function returns the direction type of tBuffer.

```
tDirectionType BufGetDirType(tBuffer buf)
```

Note: No REVERSE mode

`REVERSE` mode is not supported in the buffers anymore, although the `BufGetDirType` function is still present in the buffer interface for backwards compatibility. It will always return `DIRECT` mode.

Creating copy of buffer

This function makes a copy of the buffer.

```
UCFStatus  BufCopyBuf(tBuffer dst, tBuffer srs)
```

tBuffer dst	destination buffer
tBuffer src	source buffer

Getting the data byte length

This function returns the buffer's data length in bytes.

```
tLength  BufGetDataLen(tBuffer buf)
```

Getting the data bit length

This function returns the buffer's data length in bits.

```
tLength  BufGetDataBitLen(tBuffer buf)
```

Getting the available length

This function returns the maximal data length you can write into the buffer in one piece (in bytes).

```
tLength  BufGetAvailableLen(tBuffer buf)
```

Getting byte

This function reads one byte from the buffer. The buffer should be initialized in read mode.

```
unsigned char  BufGetByte(tBuffer buf)
```

Getting data segment

This function reads a data segment from the buffer. The buffer should be initialized in read mode.

```
unsigned char*  BufGetSeg(tBuffer buf, tLength
seglen)
```

tBuffer buf	Buffer to read from
tLength seglen	Requested data length

Peeking byte

This function peeks (reads, but does not remove) one byte from the buffer. The buffer should be initialized in read mode.

```
unsigned char  BufPeekByte(tBuffer buf)
```

Peeking data segment

This function peeks (reads, but does not remove) a data segment from the buffer. The buffer should be initialized in read mode.

```
unsigned char*  BufPeekSeg(tBuffer buf, tLength
seglen)
```

tBuffer buf	Buffer to read from
tLength seglen	Requested data length

Putting byte

This function writes one byte into the buffer. The buffer should be initialized in write mode.

```
void  BufPutByte(tBuffer buf, unsigned char byte)
```

tBuffer buf	Buffer to write into
unsigned char byte	Byte to write into buffer

Putting data segment

This function writes a data segment into the buffer. The buffer should be initialized in write mode.

```
void  BufPutSeg(tBuffer buf, unsigned char* data,
tLength seglen)
```

Putting bit

This function writes one bit into the buffer. The buffer should be initialized in write mode.

```
void BufPutBit(tBuffer bud, unsigned char bit)
```

tBuffer buf	Buffer to write into
unsigned char bit	Value of bit (0 or 1)

Getting bit

This function reads one bit from the buffer. The buffer should be initialized in read mode.

```
unsigned char BufGetBit(tBuffer buf)
```

Putting bits

This function writes bits into the buffer. The buffer should be initialized in write mode.

```
void BufPutBits(tBuffer buf, unsigned char bits,
unsigned char num)
```

tBuffer buf	Buffer to write into
unsigned char bits	Sequence of bits

unsigned char num	Number of bits in sequence (num <= 8)

Getting bits

This function reads bits from the buffer. The buffer should be initialized in read mode.

```
unsigned char BufGetBits(tBuffer buf, unsigned char
num)
```

tBuffer buf	Buffer to read from
unsigned char	Number of bits (num <= 8)

Putting padding bits

This function writes padding bits in PER ALIGN variant of encoding. The buffer should be initialized in write mode.

```
void BufPutAlign(tBuffer buf)
```

Getting padding bits

This function reads padding bits in PER ALIGN variant of encoding. The buffer should be initialized in read mode.

```
void BufGetAlign(tBuffer buf)
```

Set encoding variant (PER only)

This function sets the encoding variant.

```
void BufSetEncVar(tBuffer buf, UCFEncVariant encVar)
```

UCFEncVariant encVar	Encoding variant (UCF_Align, UCF_Unalign or UCF_NoEndPad)

Get encoding variant (PER Only)

This function returns the encoding variant.

```
UCFEncVariant  BufGetEncVar(tBuffer buf)
```

Setting up error catcher

This function sets up error catcher. Returns 0 or code of error if any.

```
unsigned int BufSetCatcher(tBuffer buf)
```

Getting error mode

This function gets error mode (bem_Off or bem_On).

```
tBufferErrorMode BufGetErrorMode(tBuffer buf)
```

Supported ASN.1 Types

- BOOLEAN
- INTEGER
- ENUMERATED
- REAL
- OBJECT IDENTIFIER
- NULL
- BIT STRING
- OCTET STRING
- IA5String
- NumericString
- PrintableString
- VisibleString
- UTCTime
- GeneralizedTime
- SEQUENCE
- SET
- SEQUENCE OF
- SET OF
- CHOICE
- Tagged types
- Open types

Encoding, and in Particular Decoding within ACM and GCI

The standard way to implement the encoder and decoder functions are to put at least one function of each in the file *encoder.c* and declare them in the file *encoder.h*. With every adaptation that is provided with the TTCN Suite installation are those files included, often with empty encode/decode functions where the user is supposed to enter the appropriate functionality. The functions should be of the following structure:

The Encoding Function

```
GciStatus Encoder_<name>(const GciValue*,
                         GciBuffer** )
```

If encoding was successful, the GciBuffer should be loaded with the encoded data and *GciOk* returned by the function. Otherwise, the function should return *GciNotOk*.

The Decoding Function

```
GciStatus Decoder_<name>(const GciBuffer*,
                        int*,
                         GciValue**)
```

If decoding was successful, the GciValue should be loaded with the decoded GCI value, the integer should contain the number of bytes that were consumed in the decoding process and the return status *GciOk*. Upon failure, the function should simply return *GciNotOk*.

Note that several encode/decode functions can exist, and the user can switch between them at will, either by calling different functions from *GciSnapshot()* or by registering different functions prior to calling *ACMSnapshot()*. See *"Registering the Active Decode Function" on page 1538 in chapter 36, Adaptation of Generated Code* for details.

General

A test suite has no knowledge of the encoding and decoding rules of the actual application protocol. The definition of signal components and the description of the signal flows are done in an abstract and high-level manner. The physical representation of the signal components and the definition of the actual transfer syntax is not defined within the test suite.

The encoding and decoding rules (functions) simply define a common transfer syntax between the test equipment and the executable test suite.

It is up to the user to write his/her own encoding and decoding rules using the GCI value representation. Even if the TTCN to C Compiler comes with an adaptation template that includes a general encoder and decoder, these rules can not be used at all times.

For test applications that need to send the same type of messages back and forth through a communication channel, the encoding and decoding

functions must be related to each other in such way that the decoding function is the inverse function of the encoding function. This gives the following simple rule:

```
Message = Decode( Encode ( Message ) )
```

This simply states, that if you decode an encoded message, you will get the original message back.

For applications that send and receive messages of different types (for example an application sending commands to an interface one way and receiving command results the other way), the encoding and decoding rules might not be related at all.

It is up to the user to identify how he/she needs to encode and decode messages to successfully be able to communicate with his/her test equipment.

Encoding and Decoding Using BER

TTCN Suite can generate encode/decode function definitions, that give an opportunity for representing values of each ASN.1 type as a string of eight-bits octets and transfer them between the environment and the ETS. TTCN Suite now have support for BER (Basic Encoding Rules) standard, defined in X.690 for the subset of ASN.1 that is defined by TTCN.

BER Encoder/Decoder Support Library

The BER support comes in the form of a static library that supports encoding/decoding functionality of ASN.1. Profiles of all the functions that a user can use are located in the `ucf.h` in the static files directory. This file should be included in the adaptation and the library should be linked together with the adaptation.

The library contains two basic functions that provide functions for encoding/decoding types, computing length of values and buffer handling procedures: `BEREncode(...)` and `BERDecode(...)`.

How to Use BER Support in the TTCN Suite?

First, select *Generate BER encoders/decoders* in the Make options dialog. A file will be generated called `asn1ende.h` in your target directory. This file defines the encoding and decoding functions for the ASN.1 definitions in the test suite.

Second, you should implement the usage of the encoding functions in your `adaptor.c`. This is done in the `GciSend` and `GciSnapshot` functions. Also, you should define buffers and initialize them (this can be done in `main`.

The generated file <ModuleName_of_ASN.1-specification>_asn1icoder.h contains the encoding/decoding function definitions for ASN.1 objects in the Test Suite. For example, if you have ASN.1 ASP Type definition by name of `ASPType1`, two functions will be generated:

```
UCFStatus Encode_ASPType1(tBuffer, GciValue*);

UCFStatus Decode_ASPType1(tBuffer, tLength*, GciValue**);
```

The first parameter of the encode function is the buffer where the encoded data is placed. The decoding is made from the buffer to a GCI value.

The encoding/decoding functions are specific for the ASP/PDU types in question so when calling them in `GciSend` and `GciSnapshot`, you need to make sure that they are called with a value of the correct ASP/PDU type. One way to get around this problem is to define a single ASP/PDU type that is a CHOICE of all types that you are sending or receiving in your test suite.

Example

For example, we have a test suite with ASN.1 ASP Type

```
ASPType1 ::= INTEGER
```

And ASN.1 ASP constraint of ASPType1 named

```
ASPCon1 ::= 5
```

We have PCO named PCO1 and dynamic behavior like this:

```
PCO1 ! ASPType1    | Constraint: ASPCon1
PCO1 ? ASPType1    | Constraint: ASPCon1
```

And we want to use BER for encoding/decoding. After code generation we have a <ModuleName_of_ASN.1-specification>_asn1icoder.h file with the following definitions:

```
tLength Encode_ASPType1(tBuffer, GciValue*);
```

```
tLength Decode_ASPType1(tBuffer, GciValue**);
```

So in our adaptation file we need to declare, initialize and close BER buffers:

```
tBuffer InBuffer;
tBuffer OutBuffer;
.
. (rest of the adaptation)
.
int main(int argc, char* argv[])
{
   BufInitBuf(InBuffer, DIRECT, status);
   BufInitBuf(OutBuffer, DIRECT, status);
   .
   . (ETS control)
   .
  BufCloseBuf(InBuffer);
  BufCloseBuf(OutBuffer);
  return 0;
}
```

In `GciSend()` we should initialize buffer in `WriteMode` and encode value:

```
GciStatus GciSend( int pcod, GciValue* object )
{
    UCFStatus status;
   BufInitWriteMode(OutBuffer);
   .
   .
  Encode_ASPType1(OutBuffer, object);
  status = Encode_ASPType1(OutBuffer, object);
  if(status != UCF_Ok)
  {
   UCFPrintErrorMessage(stderr,status);
   /* warning or exit */
  }
  .
  .
  BufCloseWriteMode(OutBuffer);
  return GciOK;
}
```

In `GciSnapshot()` we should initialize buffer in `ReadMode` and decode value:

```
GciStatus GciSnapshot( )
{
  UCFStatus status;
  tLength DecLength;
```

```
  BufInitReadMode(InBuffer);
  .
  .
  Decode_ASPType1(InBuffer, &result);
 status = Decode_ASPType1(InBuffer, &DecLength,
&result);
 if(status != UCF_Ok)
 {
  UCFPrintErrorMessage(stderr,status);
  /* warning or exit */
 }
  .
  .
  BufCloseReadMode(InBuffer);
  return GciOK;
}
```

How to Use the PER Support in the TTCN Suite

The PER support is used almost exactly like the BER support, so read above first to learn how to use the BER support. The PER Encoding/Decoding functions have been changed and now return a UCFStatus value:

```
 UCFStatus Encode_ASPType1(tBuffer, GciValue*)

 UCFStatus Decode_ASPType1(tBuffer, tLength*,
GciValue**)
```

The primary difference to the BER support is that the PER support includes supporting different variants of PER. The variants supported are Unalign, Align and NoEndPad with the Unalign variant being the default. The Unalign and Align variants are, just like BER, eight-bit octet-oriented, but the NoEndPad variant is bit oriented which necessitates use of the bit-oriented access to the buffer instead of the byte-oriented access.

The mechanism to select PER encoding variants is based on the use of pre-processor symbols. By defining the symbol `UCF_PER_DEFAULT_ENCODING_VARIANT` to the value associated with the selected encoding variant before including the `asn1ende.h` file, all types will get the selected encoding variant unless explicitly overridden.

Overriding the default encoding variant can be done on a per type basis by defining the pre-processor symbol `EncodingVariant_<type name>` to the selected variant value before including the `asn1ende.h` file.

The following example shows a snippet of the `adaptor.c` file in a situation where all types but the type `Type1` is to be encoded/decoded with the Align variant and `Type1` is to use the Unalign variant.

Example 299: Selecting PER encoding variant ––––––––––––––––––

```
#define UCF_PER_DEFAULT_ENCODING_VARIANT UCF_Align
#define EncodingVariant_Type1 UCF_Unalign
#include <asn1ende.h>
```

––

The Adaptation Framework

To make it even easier to connect the generated code to "the real world", the Adaptation Framework (ACM) is introduced, a platform-independent API that encapsulates communication and timer handling provided by *plug-in components.* These components can be communication protocol implementations, simulated timer modules, etc. and can easily be replaced *without having to rewrite the adaptation* – provided the Adaptation Framework has been used to implement the GCI functions in the adaptor.

Currently, the TTCN Suite is delivered with the following plug-in modules for the framework:

- A TCP/IP socket communication implementation.
- A system-time timer package.

This means that if the adaptation is written using the Adaptation Framework, the generated code can instantly be used with TCP/IP communication. As the framework also hides all internal mechanisms of the runtime system, the resulting adaptation will be easier to maintain, more general and much smaller in size.

Example 300: An Implementation of GciSend with ACM ––––––––––

```
GciStatus GciSend (int pcod, GciValue* msg)
{
  GciBuffer  encoded_value;
  GciStatus  status;

  encoded_value.buffer = (char *) malloc(
sizeof(char) * MAX_ENCODING_BUFFER + 1);
  encoded_value.current_length = 0;
  encoded_value.max_length = MAX_ENCODING_BUFFER;
```

```
    status = Encode(&encoded_value, msg);
    if (status != GciOk) {
      fprintf(stderr, "%s\n",
              ACMGetErrorMessage(ACMGetLastError()));
      return GciNotOk;
    }
    status = ACMSend(pcod, &encoded_value);
    free(&encoded_value);

    if (status != GciOk) {
      fprintf(stderr, "%s\n",
              ACMGetErrorMessage(ACMGetLastError()));
      return GciNotOk;
    }
    return status;
  }
```

Adaptation Templates

An empty adaptation is copied to the code directory if the user has no prior adaptation.

It is up to the user to implement the functions in the GCI operational interface. These functions are called from the TTCN runtime behavior and should not be removed even if they are empty.

The function bodies and declarations are found in the empty adaptation files, but the function bodies are empty. They only contain a print statement about the function not being implemented.

The *ACM* adaptor also uses the Adaptation Framework for communication and timer handling and is a good example to study on how to use the Adaptation Framework to implement GCI functions.

Auxiliary Adaptation Functionality

This section describes some extra functionality which is included in the adaptation templates.

`FILE* logStream;`

This is the stream to where log messages are written. The default value is **stdout** and is set in the `main` function, but can be set to whatever stream the user wishes to use.

`extern const int Gc<tablename>D = {int}`

Constant numbers for all tables (test case table, PCO table, timer table, etc...) in TTCN. Used to search in the IcSymTab array (see symbol table below). An example is for **`GcTestCaseD = 517`**.

The simple adaptation template (in the installation) includes all empty functions for the previously described GCI operational functions to be defined by the user.

Chapter

37 *Error Messages in the TTCN Suite*

This chapter contains a general description of how error messages in the TTCN Suite are structured, information about why some error messages are accompanied by a message on the standard output stream and information about how some error messages should be interpreted.

However, this chapter does not provide detailed information about specific error messages.

Error Messages

All user actions can potentially lead to an error, either because the user has made an improper action or because some system imposed limitation has been violated.

Note that all messages are not necessarily **error** messages, there are **query** and **information** messages as well.

The Structure of Error Messages

Error messages are based on information from three separate sources:

1. Tools/commands
2. TTCN Core94
3. System

The first part is always present in an error message but the other two may be inapplicable to some error messages.

Example 301 ----------

```
Open: Operation failure
Can not open the file /home/users/user/suite.itex-lock
Permission denied
```

The example above notifies you about a failed attempt to open a test suite due to you insufficient privileges to create the lock file that should automatically be created when a test suite is opened or created. Observe that as the lock file is created before any attempt is done to open the test suite, the error messages often refer to that file instead of the test suite files.

The following are examples of when the second and third parts are missing and when the third part is missing respectively.

Example 302 ----------

```
Replace: Illegal search pattern: \(foo
```

Example 303 ----------

```
Find table: Can not open foo
No such object
```

Additional Error Messages on Standard Error

The reason for additional messages on standard error in some situations, is that some functionality is accomplished by calling shell commands. If any of those shell commands fail, they write a short message to the standard error output stream. Those messages can sometimes provide additional information about why the error occurred.

Messages When Starting the TTCN Suite

When starting the TTCN Suite some error messages referring to OSF (e.g. osfBeginLine etc.) may be displayed. These messages indicate a problem in the installation related to the file XKeysymDB.

The Meaning of Error Messages

Sometimes the error messages may be a bit difficult to understand. The following list is an attempt to give some additional clues to why some errors occur and what the error messages mean:

- `Unable to display the error message directly`
  `Please check the end of the log`

  This message signifies that the system kernel do not support shared memory. The remedy is to rebuild the kernel with System V IPC support and reboot the machine. Until that is done the work-around is to look in the log for the error message.

- Messages containing `may not` or `can not` usually signifies that something is not allowed by the TTCN language. See the examples below.

Example 304 ––

```
Copy: Operation failure
May not COPY
```

––

Note:

The reason for a failed cut or copy operation may be that the selection contains items that cannot sensibly be pasted together.

Example 305 ----------

```
Rename: Can not rename Declarations Part
```

Example 306 ----------

```
May not edit name
Insert TreeHeader: Can not create tree header
```

Example 307 ----------

```
This table may not contain treeheaders
```

Chapter

38

Languages Supported in the TTCN Suite

The the TTCN Suite editing tools support TTCN and ASN.1 as defined in the TTCN standard. The TTCN to C compiler on the other hand has limitations on its ASN.1 coverage. See "TTCN ASN.1 BER Encoding/Decoding" on page 40 in chapter 1, *Compatibility Notes, in the Release Guide* for further details on the restrictions that apply.

The language supported by the TTCN Suite covers the whole of TTCN and the whole of ASN.1 as it is used in TTCN.

This chapter describes the TTCN and ASN.1 EBNF grammar supported by the TTCN Suite and the additional static semantics supported by the TTCN Suite.

The Restrictions in the TTCN Suite

The TTCN Suite editing tools support TTCN and ASN.1 as defined in the TTCN standard. The TTCN To C compiler on the other hand has limitations on its ASN.1 coverage. See "TTCN ASN.1 BER Encoding/Decoding" on page 40 in chapter 1, *Compatibility Notes, in the Release Guide* for further details on the restrictions that apply. In a few other cases, the TTCN Suite has introduced syntactical restrictions, but in all such cases there are simple work-around solutions.

This section describes the modifications in the TTCN and ASN.1 grammar, e.g. the differences between the TTCN/ASN.1 standards and the language supported by the TTCN Suite. Modifications meaning that some constructs will not be analyzed correctly even though they have legal syntax according to the TTCN/ASN.1 standards.

External ASN.1 Types

The analysis of an ASN.1 Value requires access to its Type. In TTCN it is possible to refer to an external ASN.1 Type and define an ASN.1 Value of that type.

Therefore an extra field is included in the ASN.1 type by reference tables. The user provides the definition of an external ASN.1 type in this field. The allowed syntax/semantic for this field is the same as for regular ASN.1 type definition. ASN.1 values of such a reference type will be analyzed according to the provided type definition.

The extra field is exported if the option IBM Rational MP format is chosen. It is, though, not printed together with the ASN.1 type by reference table.

Another solution to this problem is to open the referred ASN.1 module in the Organizer and define an dependencies link to it (see "Dependencies" on page 138 in chapter 2, *The Organizer* for more information). The type definition is fetched from the referred ASN.1 module before it is analyzed. For more information, see "ASN.1 External Type/Value References" on page 1197 in chapter 26, *Analyzing TTCN Documents (on UNIX)*.

ValueList

The problem is that a ValueList constraint (in MatchingSymbol) with only one element has the same construction as a parenthesized Expression in the top level. In addition, a ConstraintValue and an Expression may appear in the same places (since Expression is allowed in the ConstraintValue).

The Analyzer assumes that a ValueList with only one element is a parenthesized Expression wherever it appears.

Example 308 ––

Assuming the Test Suite Constant C1 (of type integer) with value (2 - X), C2 (also of type integer) with value 3 * (M - K) and C3 (also of type integer) with value (3, X). Both C1 and C2 are correct both syntactically and semantically, but C3 has an illegal value because value lists are not allowed in constants.

Example 309 ––

Assume a TTCN ASP Constraint C with parameters P1 and P2 (both of type integer). The parameter P1 has the value (2 + X) and P2 has the value (2 + Y, X). Both P1 and P2 have a correct value but the value of P1 is considered to be an Expression by the Analyzer.

––

ASN.1 AnyValue

This ASN.1 construct introduces ambiguities into the language. This is because it is hard to find the place where the Type ends and the Value starts.

A restricted form of AnyValue is supported where the Type in the AnyValue must be a reference to a user defined type. This form covers all cases for ASN.1 AnyValue.

Example 310 ––

```
Assume the ASN.1 Type definition T:
SEQUENCE {f1 INTEGER, f2 LT1 DEFAULT {s1 2, s2 LT2
FALSE}}
LT1 ::= SEQUENCE {s1 INTEGER, s2 ANY}
LT2 ::= BOOLEAN
```

––

ASN.1 NamedType & NamedValue

According to the ASN.1 standard (12.5), in some notation within which a type is referenced, the type may be named. In such cases, the ASN.1 standard specifies the use of the notation NamedType which is defined as:

```
NamedType ::= identifier Type | Type | SelectionType
```

The value of a type referenced using the NamedType notation shall be defined by the notation NamedValue which is defined as:

```
NamedValue ::= identifier Value | Value
```

According to the *Information Technology -OSI- ASN.1 (UDC 681.3:621.39)* the identifier in the NamedType definition shall be mandatory. The same modification is suggested for the NamedValue.

The TTCN Analyzer do not allow use of the second alternative of either of these constructs, i.e. the naming identifiers are **not** optional.

Example 311 ----------

```
Assume that the ASN.1 type T is defined as:
SEQUENCE { x INTEGER, y < Element }
Element ::= CHOICE { x BOOLEAN, y INTEGER }
This type definition is legal. The ASN.1 NamedType
in T (x & y) shall be unique (y is considered as a
NamedType in this case).
A legal value for T is: {x 2, y 4}
```

Example 312 ----------

```
Assume another ASN.1 type, T', defined as:
SEQUENCE { x INTEGER, y x < Element }
Element ::= CHOICE { x INTEGER, y BOOLEAN }
This type definition is also legal. The ASN.1
NamedType in T' (x & y) shall be unique (x is not
considered as a NamedType in this case).
The value {x 2, y 4} is not considered to be a legal
value for T' even though it is allowed according to
the ASN.1 standard. A legal value (which will be
accepted by the Analyzer) is {x 2, y x 4}
```

An important point to be noticed is that the TTCN standard does not allow the usage of an ASN.1 NamedValue within arithmetic expressions: (TTCN standard, 10.3.2.2).

Data Object Reference

A consequence of the above described restriction is that the TTCN Suite does not permit the ComponentPosition mechanism for a DataObjectReference.

Another restriction for DataObjectReference is that the TTCN Suite only permits Record referencing via ComponentIdentifiers. The mixed use of ComponentIdentifier, PDU_Identifier, and StructIdentifier for Record Reference is very vaguely described in the Standard. Almost any interpretation of the usage leads to possible ambiguities in the language.

The corresponding grammar for RecordRef (production 311) will in the TTCN Suite be:

```
RecordRef ::= Dot ComponentIdentifier
```

The “path” leading down to the data object must be fully specified, i.e. all the component identifiers from the top to the bottom must be present.

Macro Value

The wildcard values “?”, “*”, or “-” may not in the TTCN Suite be used in a Constraint where the identifier for the ASP Parameter, PDU Field, CM Element, or Structured Type Element is replaced by the macro symbol.

The Standard does not explicitly permit this nor does it allow this. The interpretation of using the wildcards for macros would however be very ambiguous. Rather than imposing one of the possible interpretations the TTCN Suite will not allow the wildcards as macro values at all.

If wildcards are to be used as values for the elements that are inserted via macro expansion they must be written in the elements of a Structured Type Constraint that is referenced as macro value.

The TTCN-MP Syntax Productions in BNF

TTCN Specification

1. TTCN_Specification ::= TTCN_Module | Package | Suite

TTCN Module

2. TTCN_Module ::= **$TTCN_Module** TTCN_ModuleId TTCN_ModuleOverviewPart [TTCN_ModuleImportPart] [DeclarationsPart] [ConstraintsPart] [DynamicPart] **$End_TTCN_Module**
3. TTCN_ModuleId ::= **$TTCN_ModuleId** TTCN_ModuleIdentifier
4. TTCN_ModuleIdentifier ::= Identifier

The Module Overview

5. TTCN_ModuleOverviewPart ::= **$TTCN_ModuleOverviewPart** TTCN_ModuleExports [TTCN_ModuleStructure] [TestCaseIndex] [TestStepIndex] [DefaultIndex] **$End_TTCN_ModuleOverviewPart**

Module Exports

6. TTCN_ModuleExports ::= **$Begin_TTCN_ModuleExports** TTCN_ModuleId [TTCN_ModuleRef] [TTCN_ModuleObjective] [StandardsRef] [PICSref] [PIXITref] [TestMethods] [Comment] ExportedObjects [Comment] **$End_ModuleExports**
7. TTCN_ModuleRef ::= **$TTCN_ModuleRef** BoundedFreeText
8. TTCN_ModuleObjective ::= **$TTCN_ModuleObjective** BoundedFreeText
9. ExportedObjects ::= **$ExportedObjects** {ExportedObject} **$End_ExportedObjects**
10. ExportedObject ::= **$ExportedObject** ObjectId ObjectType [SourceInfo] [Comment] **$End_ExportedObject**
11. ObjectId ::= **$ObjectId** ObjectIdentifier
12. ObjectIdentifier ::= Identifier | ObjectTypeReference
13. ObjectTypeReference ::= Identifier '[' Identifier ']'
14. ObjectType ::= **$ObjectType** ObjectPredefinedType

15. ObjectPredefinedType ::= **SimpleType_Object** | **StructType_Object** | **ASN1_Type_Object** | **TS_Op_Object** | **TS_Proc_Object** | **TS_Par_Object** | **SelectExpr_Object** | **TS_Const_Object** | **TS_Var_Object** | **TC_Var_Object** | **PCO_Type_Object** | **PCO_Object** | **CP_Object** | **Timer_Object** | **TComp_Object** | **TCompConfig_Object** | **TTCN_ASP_Type_Object** | **ASN1_ASP_Type_Object** | **TTCN_PDU_Type_Object** | **ASN1_PDU_Type_Object** | **TTCN_CM_Type_Object** | **ASN1_CM_Type_Object** | **EncodingRule_Object** | **EncodingVariation_Object** | **InvalidFieldEncoding_Object** | **Alias_Object** | **StructTypeConstraint_Object** | **ASN1TypeConstraint_Object** | **TTCN_ASP_Constraint_Object** | **ASN1_ASP_Constraint_Object** | **TTCN_PDU_Constraint_Object** | **ASN1_PDU_Constraint_Object** | **TTCN_CM_Constraint_Object** | **ASN1_CM_Constraint_Object** | **TestCase_Object** | **TestStep_Object** | **Default_Object** | **NamedNumber_Object** | **Enumeration_Object**
16. SourceInfo ::= **$SourceInfo** (SourceIdentifier | ObjectDirective)
17. SourceIdentifier ::= SuiteIdentifier | TTCN_ModuleIdentifier | PackageIdentifier
18. ObjectDirective ::= '-' | OMIT | EXTERNAL

TTCN Module Structure

19. TTCN_ModuleStructure ::= **$Begin_TTCN_ModuleStructure** Structure&Objectives [Comment] **$End_TTCN_ModuleStructure**

TTCN Module Import Part

20. TTCN_ModuleImportPart ::= **$TTCN_ModuleImportPart** [ExternalObjects] [ImportDeclarations] **$End_TTCN_ModuleImportPart**

External Objects

21. ExternalObjects ::= **$Begin_ExternalObjects** {ExternalObject}+ [Comment] **$End_ExternalObjects**
22. ExternalObject ::= **$ExternalObject** ExternalObjectId ObjectType [Comment] **$End_ExternalObject**
23. ExternalObjectId ::= **$ExternalObjectId** ExternalObjectIdentifier
24. ExternalObjectIdentifier ::= ObjectIdentifier | TS_OpId&ParList | ConsId&ParList | TestStepId&ParList

Import Declarations

25. ImportDeclarations ::= **$ImportDeclarations** {Imports}+ **$End_ImportDeclarations**
26. Imports ::= **$Begin_Imports** SourceId [SourceRef] [StandardsRef] [Comment] ImportedObjects [Comment] **$End_Imports**
27. SourceId ::= **$SourceId** SourceIdentifier
28. SourceRef ::= **$SourceRef** BoundedFreeText

29. ImportedObjects ::= **$ImportedObjects** {ImportedObject}+ **$End_ImportedObjects**
30. ImportedObject ::= **$ImportedObject** ObjectId ObjectType [SourceInfo] [Comment] **$End_ImportedObject**

Source Package

31. Package ::= **$Package** PackageId PackageExports PackageImports **$End_Package**
32. PackageId ::= **$PackageId** PackageIdentifier
33. PackageIdentifier ::= Identifier

Package exports

34. PackageExports ::= **$Begin_PackageExports** PackageId [Comment] PackageExportedObjects [Comment] **$End_PackageExports**
35. PackageExportedObjects ::= **$PackageExportedObjects** {PackageExportedObject}+ **$End_PackageExportedObjects**
36. PackageExportedObject ::= **$PackageExportedObject** ObjectId ObjectType [SourceInfo] [Comment] **$End_PackageExportedObject**

Package imports and renames

37. PackageImports ::= **$PackageImports** {PackageImport} **$End_PackageImports**

Package Import

38. PackageImport ::= **$Begin_PackageImport** SourceId [Comment] PackageImportedObjects [Comment] **$End_PackageImport**
39. PackageImportedObjects ::= **$PackageImportedObjects** {PackageImportedObject}+ **$End_PackageImportedObjects**
40. PackageImportObject ::= **$PackageImportObject** ObjectId ObjectType [NewObjectDef] [NewObjectId] [Comment] **$End_PackageImportObject**
41. NewObjectDef ::= **$NewObjectDef** (NewObjectIdentifier | ObjectDirective)
42. NewObjectId ::= **$NewObjectId** NewObjectIdentifier
43. NewObjectIdentifier ::= ObjectIdentifier

Test suite

44. Suite ::= **$Suite** SuiteId SuiteOverviewPart [ImportPart] DeclarationsPart ConstraintsPart DynamicPart **$End_Suite**
45. SuiteId ::= **$SuiteId** SuiteIdentifier
46. SuiteIdentifier ::= Identifier

The Test Suite Overview

47. SuiteOverviewPart ::= **$SuiteOverviewPart** [TestSuiteIndex] SuiteStructure [TestCaseIndex] [TestStepIndex] [DefaultIndex] **$End_SuiteOverviewPart**

Test Suite Index

48. TestSuiteIndex ::= **$Begin_TestSuiteIndex** {ObjectInfo} [Comment] **$End_TestSuiteIndex**

The Imported Object Info

49. ObjectInfo ::= **$ObjectInfo** ObjectId ObjectType SourceId OrigObjectId [PageNum] [Comment] **$End_ObjectInfo**
50. PageNum ::= **$PageNum** PageNumber
51. PageNumber ::= Number
52. OrigObjectId ::= **$OrigObjectId** ObjectIdentifier

Test Suite Structure

53. SuiteStructure ::= **$Begin_SuiteStructure** SuiteId StandardsRef PICSref PIXITref TestMethods [Comment] Structure&Objectives [Comment] **$End_SuiteStructure**
54. StandardsRef ::= **$StandardsRef** BoundedFreeText
55. PICSref ::= **$PICSref** BoundedFreeText
56. PIXITref ::= **$PIXITref** BoundedFreeText
57. TestMethods ::= **$TestMethods** BoundedFreeText
58. Comment ::= **$Comment** [BoundedFreeText]
59. Structure&Objectives ::= **$Structure&Objectives** {Structure&Objective} **$End_Structure&Objectives**
60. Structure&Objective ::= **$Structure&Objective** TestGroupRef SelExprId Objective **$End_Structure&Objective**
61. SelExprId ::= **$SelectExprId** [SelectExprIdentifier]

Test Case Index

62. TestCaseIndex ::= **$Begin_TestCaseIndex** {CaseIndex} [Comment] **$End_TestCaseIndex**
63. CaseIndex ::= **$CaseIndex** TestGroupRef TestCaseId SelExprId Description **$End_CaseIndex**
64. Description ::= **$Description** BoundedFreeText

Test Step Index

65. TestStepIndex ::= **$Begin_TestStepIndex** {StepIndex} [Comment]
    $End_TestStepIndex
66. StepIndex ::= **$StepIndex** TestStepRef TestStepId Description **$End_StepIndex**

Default Index

67. DefaultIndex ::= **$Begin_DefaultIndex** {DefIndex} [Comment]
    $End_DefaultIndex
68. DefIndex ::= **$DefIndex** DefaultRef DefaultId Description **$End_DefIndex**

The Import Part

69. ImportPart ::= **$ImportPart** ImportDeclarations **$End_ImportPart**

The Declarations Part

70. DeclarationsPart ::= **$DeclarationsPart** Definitions Parameterization&Selection
    Declarations ComplexDefinitions **$End_DeclarationsPart**

Definitions

71. Definitions ::= [TS_TypeDefs] [EncodingDefs] [TS_OpDefs] [TS_ProcDefs]

Test Suite Type Definitions

72. TS_TypeDefs ::= **$TS_TypeDefs** [SimpleTypeDefs] [StructTypeDefs]
    [ASN1_TypeDefs] [ASN1_TypeRefs] **$End_TS_TypeDefs**

Simple Type Definitions

73. SimpleTypeDefs ::= **$Begin_SimpleTypeDefs** {SimpleTypeDef}+ [Comment] **$End_SimpleTypeDefs**
74. SimpleTypeDef ::= **$SimpleTypeDef** SimpleTypeId SimpleTypeDefinition [PDU_FieldEncoding] [Comment] **$End_SimpleTypeDef**
75. SimpleTypeId ::= **$SimpleTypeId** SimpleTypeIdentifier
76. SimpleTypeIdentifier ::= Identifier
77. SimpleTypeDefinition ::= **$SimpleTypeDefinition** Type&Restriction
78. Type&Restriction ::= Type [Restriction]
79. Restriction ::= LengthRestriction | IntegerRange | SimpleValueList
80. LengthRestriction ::= SingleTypeLength | RangeTypeLength
81. SingleTypeLength ::= "["Number "]"
82. RangeTypeLength ::= "[" LowerTypeBound To UpperTypeBound "]"
83. IntegerRange ::= "(" LowerTypeBound To UpperTypeBound ")"
84. LowerTypeBound ::= [Minus] Number | Minus **INFINITY**
85. UpperTypeBound ::= [Minus] Number | **INFINITY**
86. To ::= **TO** | ".."
87. SimpleValueList ::= "(" [Minus] LiteralValue {Comma [Minus] LiteralValue} ")"

Structured Type Definitions

88. StructTypeDefs ::= **$StructTypeDefs** {StructTypeDef}+ **$End_StructTypeDefs**
89. StructTypeDef ::= **$Begin_StructTypeDef** StructId [EncVariationId] [Comment] ElemDcls [Comment] **$End_StructTypeDef**
90. StructId ::= **$StructId** StructId&FullId
91. StructId&FullId ::= StructIdentifier [FullIdentifier]
92. FullIdentifier ::= "(" BoundedFreeText ")"
93. StructIdentifier ::= Identifier
94. ElemDcls ::= **$ElemDcls** {ElemDcl}+ **$End_ElemDcls**
95. ElemDcl ::= **$ElemDcl** ElemId ElemType [PDU_FieldEncoding] [Comment] **$End_ElemDcl**
96. ElemId ::= **$ElemId** ElemId&FullId
97. ElemId&FullId ::= ElemIdentifier [FullIdentifier]
98. ElemIdentifier ::= Identifier
99. ElemType ::= **$ElemType** Type&Attributes

ASN.1 Type Definitions

100.ASN1_TypeDefs ::= **$ASN1_TypeDefs** {ASN1_TypeDef}+
$End_ASN1_TypeDefs
101.ASN1_TypeDef ::= **$Begin_ASN1_TypeDef** ASN1_TypeId [EncVariationId]
[Comment] ASN1_TypeDefinition [Comment] **$End_ASN1_TypeDef**
102.ASN1_TypeId ::= **$ASN1_TypeId** ASN1_TypeId&FullId
103.ASN1_TypeId&FullId ::= ASN1_TypeIdentifier [FullIdentifier]
104.ASN1_TypeIdentifier ::= Identifier
105.ASN1_TypeDefinition ::= **$ASN1_TypeDefinition** ASN1_Type&LocalTypes
$End_ASN1_TypeDefinition
106.ASN1_Type&LocalTypes ::= ASN1_Type {ASN1_LocalType}
107.ASN1_Type ::= ASN1_main_Type [ASN1_Encoding]
108.ASN1_LocalType ::= TypeAssignment

ASN.1 Type Definitions by Reference

109.ASN1_TypeRefs ::= **$Begin_ASN1_TypeRefs** {ASN1_TypeRef}+ [Comment]
$End_ASN1_TypeRefs
110.ASN1_TypeRef ::= **$ASN1_TypeRef** ASN1_TypeId ASN1_TypeReference
ASN1_ModuleId [EncVariationId] [Comment] [*ASN1_TypeDefinition*]
$End_ASN1_TypeRef
111.ASN1_TypeReference ::= **$ASN1_TypeReference** TypeReference
112.TypeReference ::= typereference
113.ASN1_ModuleId ::= **$ASN1_ModuleId** ModuleIdentifier
114.ModuleIdentifier ::= moduleidentifier

Test Suite Operation Definitions

115.TS_OpDefs ::= **$TS_OpDefs** {TS_OpDef}+ **$End_TS_OpDefs**
116.TS_OpDef ::= **$Begin_TS_OpDef** TS_OpId TS_OpResult [Comment]
TS_OpDescription [Comment] **$End_TS_OpDef**
117.TS_OpId ::= **$TS_OpId** TS_OpId&ParList
118.TS_OpId&ParList ::= TS_OpIdentifier [FormalParList]
119.TS_OpIdentifier ::= Identifier
120.TS_OpResult ::= **$TS_OpResult** TypeOrPDU
121.TS_OpDescription ::= **$TS_OpDescription** BoundedFreeText

Test Suite Operation Procedural Definitions

122.TS_ProcDefs ::= **$TS_ProcDefs** {TS_ProcDef}+ **$End_TS_ProcDefs**
123.TS_ProcDef ::= **$Begin_TS_ProcDef** TS_ProcId TS_ProcResult [Comment] TS_ProcDescription [Comment] **$End_TS_ProcDef**
124.TS_ProcId ::= **$TS_ProcId** TS_ProcId&ParList
125.TS_ProcId&ParList ::= TS_ProcIdentifier [FormalParList]
126.TS_ProcIdentifier ::= Identifier
127.TS_ProcResult ::= **$TS_ProcResult** TypeOrPDU
128.TS_ProcDescription ::= **$TS_ProcDescription** TS_OpProcDef **$End_TS_ProcDescription**
129.TS_OpProcDef ::= [VarBlock] ProcStatement
130.VarBlock ::= **VAR** VarDcls **ENDVAR**
131.VarDcls ::= {VarDcl SemiColon}
132.VarDcl ::= [**STATIC**] VarIdentifiers Colon TypeOrPDU [Colon Value]
133.VarIdentifiers ::= VarIdentifier {Comma VarIdentifier}
134.VarIdentifier ::= Identifier
135.ProcStatement ::= ReturnValueStatement | Assignment | IfStatement | WhileLoop | CaseStatement | ProcBlock
136.ReturnValueStatement ::= **RETURNVALUE** Expression
137.IfStatement ::= **IF** Expression **THEN** {ProcStatement SemiColon}+ [**ELSE** {ProcStatement SemiColon}+] **ENDIF**
138.WhileLoop ::= **WHILE** Expression **DO** {ProcStatement SemiColon}+ **END-WHILE**
139.CaseStatement ::= **CASE** Expression **OF** {CaseClause SemiColon}+ [**ELSE** {ProcStatement SemiColon}+] **ENDCASE**
140.CaseClause ::= IntegerLabel Colon ProcStatement
141.IntegerLabel ::= Number | TS_ParIdentifier | TS_ConstIdentifier
142.ProcBlock ::= **BEGIN** {ProcStatement SemiColon}+ **END**

Parameterization and Selection

143.Parameterization&Selection ::= [TS_ParDcls] [SelectExprDefs]

Test Suite Parameter Declarations

144.TS_ParDcls ::= **$Begin_TS_ParDcls** {TS_ParDcl}+ [Comment]
$End_TS_ParDcls
145.TS_ParDcl ::= **$TS_ParDcl** TS_ParId TS_ParType PICS_PIXITref [Comment]
$End_TS_ParDcl
146.TS_ParId ::= **$TS_ParId** TS_ParIdentifier
147.TS_ParIdentifier ::= Identifier
148.TS_ParType ::= **$TS_ParType** TypeOrPDU
149.PICS_PIXITref ::= **$PICS_PIXITref** BoundedFreeText

Test Case Selection Expression Definitions

150.SelectExprDefs ::= **$Begin_SelectExprDefs** {SelectExprDef}+ [Comment]
$End_SelectExprDefs
151.SelectExprDef ::= **$SelectExprDef** SelectExprId SelectExpr [Comment]
$End_SelectExprDef
152.SelectExprId ::= **$SelectExprId** SelectExprIdentifier
153.SelectExprIdentifier ::= Identifier
154.SelectExpr ::= **$SelectExpr** SelectionExpression
155.SelectionExpression ::= Expression

Declarations

156.Declarations ::= [TS_ConstDcls] [TS_ConstRefs] [TS_VarDcls] [TC_VarDcls] [PCO_TypeDcls] [PCO_Dcls] [CP_Dcls] [TimerDcls] [TCompDcls TCompConfigDcls]

Test Suite Constant Declarations

157.TS_ConstDcls ::= **$Begin_TS_ConstDcls** {TS_ConstDcl}+ [Comment]
$End_TS_ConstDcls
158.TS_ConstDcl ::= **$TS_ConstDcl** TS_ConstId TS_ConstType TS_ConstValue [Comment] **$End_TS_ConstDcl**
159.TS_ConstId ::= **$TS_ConstId** TS_ConstIdentifier
160.TS_ConstIdentifier ::= Identifier
161.TS_ConstType ::= **$TS_ConstType** Type
162.TS_ConstValue ::= **$TS_ConstValue** DeclarationValue
163.DeclarationValue ::= Expression

Test Suite Constant Declarations by Reference

164.TS_ConstRefs ::= **$Begin_TS_ConstRefs** {TS_ConstRef}+ [Comment]
$End_TS_ConstRefs
165.TS_ConstRef ::= **$TS_ConstRef** TS_ConstId TS_ConstType
ASN1_ValueReference ASN1_ModuleId [Comment] *[ASN1_Value]*
$End_TS_ConstRef
166.ASN1_ValueReference ::= **$ASN1_ValueReference** ValueReference
167.ValueReference ::= valuereference

Test Suite Variable Declarations

168.TS_VarDcls ::= **$Begin_TS_VarDcls** {TS_VarDcl}+ [Comment]
$End_TS_VarDcls
169.TS_VarDcl ::= **$TS_VarDcl** TS_VarId TS_VarType TS_VarValue [Comment]
$End_TS_VarDcl
170.TS_VarId ::= **$TS_VarId** TS_VarIdentifier
171.TS_VarIdentifier ::= Identifier
172.TS_VarType ::= **$TS_VarType** TypeOrPDU
173.TS_VarValue ::= **$TS_VarValue** [DeclarationValue]

Test Case Variable Declarations

174.TC_VarDcls ::= **$Begin_TC_VarDcls** {TC_VarDcl}+ [Comment]
$End_TC_VarDcls
175.TC_VarDcl ::= **$TC_VarDcl** TC_VarId TC_VarType TC_VarValue [Comment]
$End_TC_VarDcl
176.TC_VarId ::= **$TC_VarId** TC_VarIdentifier
177.TC_VarIdentifier ::= Identifier
178.TC_VarType ::= **$TC_VarType** TypeOrPDU
179.TC_VarValue ::= **$TC_VarValue** [DeclarationValue]

PCO Type Declarations

180.PCO_TypeDcls ::= **$Begin_PCO_TypeDcls** {PCO_TypeDcl}+ [Comment] **$End_PCO_TypeDcls**
181.PCO_TypeDcl ::= **$PCO_TypeDcl** PCO_TypeId P_Role [Comment] **$End_PCO_TypeDcl**
182.PCO_TypeId ::= **$PCO_TypeId** PCO_TypeIdentifier
183.PCO_TypeIdentifier ::= Identifier

PCO Declarations

184.PCO_Dcls ::= **$Begin_PCO_Dcls** {PCO_Dcl}+ [Comment] **$End_PCO_Dcls**
185.PCO_Dcl ::= **$PCO_Dcl** PCO_Id PCO_TypeId&MuxValue P_Role [Comment] **$End_PCO_Dcl**
186.PCO_Id ::= **$PCO_Id** PCO_Identifier
187.PCO_Identifier ::= Identifier
188.PCO_TypeId&MuxValue ::= **$PCO_TypeId** PCO_TypeIdentifier [“(“ MuxValue “)”]
189.MuxValue ::= TS_ParIdentifier
190.P_Role ::= **$PCO_Role** PCO_Role
191.PCO_Role ::= **UT** | **LT**

Coordination Points Declaration

192.CP_Dcls ::= **$Begin_CP_Dcls** {CP_Dcl}+ [Comment] **$End_CP_Dcls**
193.CP_Dcl ::= **$CP_Dcl** CP_Id [Comment] **$End_CP_Dcl**
194.CP_Id ::= **$CP_Id** CP_Identifier
195.CP_Identifier ::= Identifier

Timer Declarations

196.TimerDcls ::= **$Begin_TimerDcls** {TimerDcl}+ [Comment] **$End_TimerDcls**
197.TimerDcl ::= **$TimerDcl** TimerId Duration Unit [Comment] **$End_TimerDcl**
198.TimerId ::= **$TimerId** TimerIdentifier
199.TimerIdentifier ::= Identifier
200.Duration ::= **$Duration** [DeclarationValue]
201.Unit ::= **$Unit** TimeUnit
202.TimeUnit ::= **ps** | **ns** | **us** | **ms** | **s** | **min**

Test Component Declarations

203.TCompDcls ::= **$Begin_TCompDcls** {TCompDcl}+ [Comment] **$End_TCompDcls**
204.TCompDcl ::= **$TCompDcl** TCompId TCompRole NumOf PCOs NumOf CPs [Comment] **$EndTCompDcl**
205.TCompId ::= **$TCompId** TCompIdentifier
206.TCompIdentifier ::= Identifier
207.TCompRole ::= **$TCompRole** TC_Role
208.TC_Role ::= **MTC** | **PTC**
209.NumOf_PCOs ::= **$NumOf_PCOs** Num_PCOs
210.Num_PCOs ::= Number
211.NumOf_Cps ::= **$NumOf_CPs** Num_CPs
212.Num_CPs ::= Number

Test Component Configuration Declarations

213.TCompConfigDcls ::= **$TCompConfigDcls** {TCompConfigDcl}+ **$End_TCompConfigDcls**
214.TCompConfigDcl ::= **$Begin_TCompConfigDcl** TCompConfigId [Comment] TCompConfigInfos [Comment] **$EndTCompConfigDcl**
215.TCompConfigId ::= **$TCompConfigId** TCompConfigIdentifier
216.TCompConfigIdentifier ::= Identifier
217.TCompConfigInfos ::= **$TCompConfigInfos** {TCompConfigInfo}+ **$End_TCompConfigInfos**
218.TCompConfigInfo ::= **$TCompConfigInfo** TCompUsed PCOs_Used CPs_Used [Comment] **$End_TCompConfigInfo**
219.TCompUsed ::= **$TCompUsed** TCompIdentifier
220.PCOs_Used ::= **$PCOs_Used** [PCO_List]
221.PCO_List ::= PCO_Identifier {Comma PCO_Identifier}
222.CPs_Used ::= **$CPs_Used** [CP_List]
223.CP_List ::= CP_Identifier {Comma CP_Identifier}

ASP, PDU and CM Type Definitions

224.ComplexDefinitions ::= [ASP_TypeDefs] [PDU_TypeDefs] [CM_TypeDefs] [AliasDefs]

ASP Type Definitions

225.ASP_TypeDefs ::= **$ASP_TypeDefs** [TTCN_ASP_TypeDefs] [ASN1_ASP_TypeDefs] [ASN1_ASP_TypeDefsByRef] **$End_ASP_TypeDefs**

Tabular ASP Type Definitions

226.TTCN_ASP_TypeDefs ::= **$TTCN_ASP_TypeDefs** {TTCN_ASP_TypeDef}+ **$End_TTCN_ASP_TypeDefs**
227.TTCN_ASP_TypeDef ::= **$Begin_TTCN_ASP_TypeDef** ASP_Id PCO_Type [Comment] [ASP_ParDcls] [Comment] **$End_TTCN_ASP_TypeDef**
228.PCO_Type ::= **$PCO_Type** [PCO_TypeIdentifier]
229.ASP_Id ::= **$ASP_Id** ASP_Id&FullId
230.ASP_Id&FullId ::= ASP_Identifier [FullIdentifier]
231.ASP_Identifier ::= Identifier
232.ASP_ParDcls ::= **$ASP_ParDcls** {ASP_ParDcl} **$End_ASP_ParDcls**
233.ASP_ParDcl ::= **$ASP_ParDcl** ASP_ParId ASP_ParType [Comment] **$End_ASP_ParDcl**
234.ASP_ParId ::= **$ASP_ParId** ASP_ParIdOrMacro
235.ASP_ParIdOrMacro ::= ASP_ParId&FullId | MacroSymbol
236.ASP_ParId&FullId ::= ASP_ParIdentifier [FullIdentifier]
237.ASP_ParIdentifier ::= Identifier
238.ASP_ParType ::= **$ASP_ParType** Type&Attributes

ASN.1 ASP Type Definitions

239.ASN1_ASP_TypeDefs ::= **$ASN1_ASP_TypeDefs** {ASN1_ASP_TypeDef} **$End_ASN1_ASP_TypeDefs**
240.ASN1_ASP_TypeDef ::= **$Begin_ASN1_ASP_TypeDef** ASP_Id PCO_Type [Comment] [ASN1_TypeDefinition] [Comment] **$End_ASN1_ASP_TypeDef**

ASN.1 ASP Type Definitions by Reference

241.ASN1_ASP_TypeDefsByRef ::= **$Begin_ASN1_ASP_TypeDefsByRef** {ASN1_ASP_TypeDefByRef}+ [Comment] **$End_ASN1_ASP_TypeDefsByRef**
242.ASN1_ASP_TypeDefByRef ::= **$ASN1_ASP_TypeDefByRef** ASP_Id PCO_Type ASN1_TypeReference ASN1_ModuleId [Comment] *[ASN1_TypeDefinition]* **$End_ASN1_ASP_TypeDefByRef**

PDU Type Definitions

243.PDU_TypeDefs ::= **$PDU_TypeDefs** [TTCN_PDU_TypeDefs] [ASN1_PDU_TypeDefs] [ASN1_PDU_TypeDefsByRef] **$End_PDU_TypeDefs**

Tabular PDU Type Definitions

244.TTCN_PDU_TypeDefs ::= **$TTCN_PDU_TypeDefs** {TTCN_PDU_TypeDef}+ **$End_TTCN_PDU_TypeDefs**
245.TTCN_PDU_TypeDef ::= **$Begin_TTCN_PDU_TypeDef** PDU_Id PCO_Type [PDU_EncodingId] [EncVariationId] [Comment] [PDU_FieldDcls] [Comment] **$End_TTCN_PDU_TypeDef**
246.PDU_Id ::= **$PDU_Id** PDU_Id&FullId
247.PDU_Id&FullId ::= PDU_Identifier [FullIdentifier]
248.PDU_Identifier ::= Identifier
249.PDU_EncodingId ::= **$PDU_EncodingId** [EncodingRuleIdentifier]
250.PDU_FieldDcls ::= **$PDU_FieldDcls** {PDU_FieldDcl} **$End_PDU_FieldDcls**
251.PDU_FieldDcl ::= **$PDU_FieldDcl** PDU_FieldId PDU_FieldType [PDU_FieldEncoding] [Comment] **$End_PDU_FieldDcl**
252.PDU_FieldId ::= **$PDU_FieldId** PDU_FieldIdOrMacro
253.PDU_FieldIdOrMacro ::= PDU_FieldId&FullId | MacroSymbol
254.MacroSymbol ::= "<-"
255.PDU_FieldId&FullId ::= PDU_FieldIdentifier [FullIdentifier]
256.PDU_FieldIdentifier ::= Identifier
257.PDU_FieldType ::= **$PDU_FieldType** Type&Attributes
258.Type&Attributes ::= (Type [LengthAttribute]) | **PDU**
259.LengthAttribute ::= SingleLength | RangeLength
260.SingleLength ::= "[" Bound "]"
261.Bound ::= Number | TS_ParIdentifier | TS_ConstIdentifier
262.RangeLength ::= "[" LowerBound To UpperBound "]"
263.LowerBound ::= Bound
264.UpperBound ::= Bound | **INFINITY**

ASN.1 PDU Type Definitions

265.ASN1_PDU_TypeDefs ::= **$ASN1_PDU_TypeDefs** {ASN1_PDU_TypeDef} **$End_ASN1_PDU_TypeDefs**
266.ASN1_PDU_TypeDef ::= **$Begin_ASN1_PDU_TypeDef** PDU_Id PCO_Type [PDU_EncodingId] [EncVariationId] [Comment] [ASN1_TypeDefinition] [Comment] **$End_ASN1_PDU_TypeDef**

ASN.1 PDU Type Definitions by Reference

267.ASN1_PDU_TypeDefsByRef ::= **$Begin_ASN1_PDU_TypeDefsByRef** {ASN1_PDU_TypeDefByRef}+ [Comment] **$End_ASN1_PDU_TypeDefsByRef**

268.ASN1_PDU_TypeDefByRef ::= **$ASN1_PDU_TypeDefByRef** PDU_Id PCO_Type ASN1_TypeReference ASN1_ModuleId [PDU_EncodingId] [Enc-VariationId] [Comment] *[ASN1_TypeDefinition]* **$End_ASN1_PDU_TypeDefByRef**

CM Type Definitions

269.CM_TypeDefs ::= **$CM_TypeDefs** [TTCN_CM_TypeDefs] [ASN1_CM_TypeDefs] **$End_CM_TypeDefs**

Tabular CM Type Definitions

270.TTCN_CM_TypeDefs ::= **$TTCN_CM_TypeDefs** {TTCN_CM_TypeDefs}+ **$End_TTCN_CM_TypeDefs**

271.TTCN_CM_TypeDef ::= **$Begin_TTCN_CM_TypeDef** CM_Id [Comment] [CM_ParDcls] [Comment] **$End_TTCN_CM_TypeDef**

272.CM_Id ::= **$CM_Id** CM_Identifier

273.CM_Identifier ::= Identifier

274.CM_ParDcls ::= **$CM_ParDcls** {CM_ParDcl} **$End_CM_ParDcls**

275.CM_ParDcl ::= **$CM_ParDcl** CM_ParId CM_ParType [Comment] **$End_CM_ParDcl**

276.CM_ParId ::= **$CM_ParId** CM_ParIdOrMacro

277.CM_ParIdOrMacro ::= CM_ParIdentifier | MacroSymbol

278.CM_ParIdentifier ::= Identifier

279.CM_ParType ::= **$CM_ParType** Type&Attributes

ASN1 CM Type Definitions

280.ASN1_CM_TypeDefs ::= **$ASN1_CM_TypeDefs** {ASN1_CM_TypeDefs}+ **$End_ASN1_CM_TypeDefs**

281.ASN1_CM_TypeDef ::= **$Begin_ASN1_CM_TypeDef** CM_Id [Comment] [ASN1_TypeDefinition] [Comment] **$End_ASN1_CM_TypeDef**

Varieties of Encoding Definition

282.EncodingDefs ::= **$EncodingDefs** [EncodingDefinitions] [EncodingVariations] [InvalidFieldEncodingDefs] **$End_EncodingDefs**

Encoding Definitions

283.EncodingDefinitions ::= **$Begin_EncodingDefinitions** {EncodingDefinition}+ [Comment] **$End_EncodingDefinitions**
284.EncodingDefinition ::= **$EncodingDefinition** EncodingRuleId EncodingRef EncodingDefault [Comment] **$End_EncodingDefinition**
285.EncodingRuleId ::= **$EncodingRuleId** EncodingRuleIdentifier
286.EncodingRuleIdentifier ::= Identifier
287.EncodingRef ::= **$EncodingRef** EncodingReference
288.EncodingReference ::= BoundedFreeText
289.EncodingDefault ::= **$EncodingDefault** [DefaultExpression]
290.DefaultExpression ::= Expression

Encoding Variations

291.EncodingVariations ::= **$EncodingVariations** {EncodingVariationSet}+ **$End_EncodingVariations**
292.EncodingVariationSet ::= **$Begin_EncodingVariationSet** EncodingRuleId Encoding_TypeList [Comment] $EncodingVariationList {EncodingVariation}+ $End_EncodingVariationList [Comment] **$End_EncodingVariationSet**
293.Encoding_TypeList ::= **$Encoding_TypeList** [TypeList]
294.TypeList ::= Type {Comma Type}
295.EncodingVariation ::= **$EncodingVariation** EncodingVariationId VariationRef VariationDefault [Comment] **$End_EncodingVariation**
296.EncodingVariationId ::= **$EncodingVariationId** EncVariationId&ParList
297.EncVariationId&ParList ::= EncVariationIdentifier [FormalParList]
298.EncVariationIdentifier ::= Identifier
299.VariationRef ::= **$VariationRef** VariationReference
300.VariationReference ::= BoundedFreeText
301.VariationDefault ::= **$VariationDefault** [DefaultExpression]

Invalid Encoding Definitions

302.InvalidFieldEncodingDefs ::= **$InvalidFieldEncodingDefs** {InvalidFieldEncodingDef}+ **$End_InvalidFieldEncodingDefs**
303.InvalidFieldEncodingDef ::= **$Begin_InvalidFieldEncodingDef** InvalidFieldEncodingId Encoding_TypeList [Comment] InvalidFieldEncodingDefinition [Comment] **$End_InvalidFieldEncodingDef**
304.InvalidFieldEncodingId ::= **$InvalidFieldEncodingId** InvalidFieldEncodingId&ParList

305.InvalidFieldEncodingId&ParList ::= InvalidFieldEncodingIdentifier [FormalParList]
306.InvalidFieldEncodingIdentifier ::= Identifier
307.InvalidFieldEncodingDefinition ::= **$InvalidFieldEncodingDefinition** TS_OpProcDef **$End_InvalidFieldEncodingDefinition**

Alias Definitions

308.AliasDefs ::= **$Begin_AliasDefs** {AliasDef}+ [Comment] **$End_AliasDefs**
309.AliasDef ::= **$AliasDef** AliasId ExpandedId [Comment] **$End_AliasDef**
310.AliasId ::= **$AliasId** AliasIdentifier
311.AliasIdentifier ::= Identifier
312.ExpandedId ::= **$ExpandedId** Expansion
313.Expansion ::= ASP_Identifier | PDU_Identifier

The Constraints Part

314.ConstraintsPart ::= **$ConstraintsPart** [TS_TypeConstraints] [ASP_Constraints] [PDU_Constraints] [CM_Constraints] **$End_ConstraintsPart**

Test Suite Type Constraint Declarations

315.TS_TypeConstraints ::= **$TS_TypeConstraints** [StructTypeConstraints] [ASN1_TypeConstraints] **$End_TS_TypeConstraints**

Structured Type Constraint Declarations

316.StructTypeConstraints ::= **$StructTypeConstraints** {StructTypeConstraint}+ **$End_StructTypeConstraints**
317.StructTypeConstraint ::= **$Begin_StructTypeConstraint** ConsId StructId DerivPath [EncVariationId] [Comment] ElemValues [Comment] **$End_StructTypeConstraint**
318.EncVariationId ::= **$EncVariationId** [EncVariationCall]
319.EncVariationCall ::= EncVariationIdentifier [ActualParList]
320.ElemValues ::= **$ElemValues** {ElemValue}+ **$End_ElemValues**
321.ElemValue ::= **$ElemValue** ElemId ConsValue [PDU_FieldEncoding] [Comment] **$End_ElemValue**
322.PDU_FieldEncoding ::= **$PDU_FieldEncoding** [PDU_FieldEncodingCall]
323.PDU_FieldEncodingCall ::= EncVariationCall | InvalidFieldEncodingCall
324.InvalidFieldEncodingCall ::= InvalidFieldEncodingIdentifier (ActualParList | "(" ")")

ASN.1 Type Constraint Declarations

325.ASN1_TypeConstraints ::= **$ASN1_TypeConstraints** {ASN1_TypeConstraint}+ **$End_ASN1_TypeConstraints**
326.ASN1_TypeConstraint ::= **$Begin_ASN1_TypeConstraint** ConsId ASN1_TypeId DerivPath [EncVariationId] [Comment] ASN1_ConsValue [Comment] **$End_ASN1_TypeConstraint**

ASP Constraint Declarations

327.ASP_Constraints ::= **$ASP_Constraints** [TTCN_ASP_Constraints] [ASN1_ASP_Constraints] **$End_ASP_Constraints**

Tabular ASP Constraint Declarations

328.TTCN_ASP_Constraints ::= **$TTCN_ASP_Constraints** {TTCN_ASP_Constraint}+ **$End_TTCN_ASP_Constraints**
329.TTCN_ASP_Constraint ::= **$Begin_TTCN_ASP_Constraint** ConsId ASP_Id DerivPath [Comment] [ASP_ParValues] [Comment] **$End_TTCN_ASP_Constraint**
330.ASP_ParValues ::= **$ASP_ParValues** {ASP_ParValue}+ **$End_ASP_ParValues**
331.ASP_ParValue ::= **$ASP_ParValue** ASP_ParId ConsValue [Comment] **$End_ASP_ParValue**

ASN.1 ASP Constraint Declarations

332.ASN1_ASP_Constraints ::= **$ASN1_ASP_Constraints** {ASN1_ASP_Constraint}+ **$End_ASN1_ASP_Constraints**
333.ASN1_ASP_Constraint ::= **$Begin_ASN1_ASP_Constraint** ConsId ASP_Id DerivPath [Comment] [ASN1_ConsValue] [Comment] **$End_ASN1_ASP_Constraint**

PDU Constraint Declarations

334.PDU_Constraints ::= **$PDU_Constraints** [TTCN_PDU_Constraints] [ASN1_PDU_Constraints] **$End_PDU_Constraints**

Tabular PDU Constraint Declarations

335.TTCN_PDU_Constraints ::= **$TTCN_PDU_Constraints** {TTCN_PDU_Constraint}+ **$End_TTCN_PDU_Constraints**
336.TTCN_PDU_Constraint ::= **$Begin_TTCN_PDU_Constraint** ConsId PDU_Id DerivPath [EncRuleId] [EncVariationId] [Comment] [PDU_FieldValues] [Comment] **$End_TTCN_PDU_Constraint**
337.EncRuleId ::= **$EncRuleId** [EncodingRuleIdentifier]
338.ConsId ::= **$ConsId** ConsId&ParList
339.ConsId&ParList ::= ConstraintIdentifier [FormalParList]
340.ConstraintIdentifier ::= Identifier
341.DerivPath ::= **$DerivPath** [DerivationPath]
342.DerivationPath ::= {ConstraintIdentifier Dot}+
343.PDU_FieldValues ::= **$PDU_FieldValues** {PDU_FieldValue}+ **$End_PDU_FieldValues**
344.PDU_FieldValue ::= **$PDU_FieldValue** PDU_FieldId ConsValue [PDU_FieldEncoding] [Comment] **$End_PDU_FieldValue**
345.ConsValue ::= **$ConsValue** ConstraintValue&Attributes
346.ConstraintValue&Attributes ::= ConstraintValue ValueAttributes
347.ConstraintValue ::= ConstraintExpression | MatchingSymbol | ConsRef
348.ConstraintExpression ::= Expression
349.MatchingSymbol ::= Complement | Omit | AnyValue | AnyOrOmit | ValueList | ValueRange | SuperSet | SubSet | Permutation
350.Complement ::= **COMPLEMENT** ValueList
351.Omit ::= Dash | **OMIT**
352.AnyValue ::= "?"
353.AnyOrOmit ::= "*"
354.ValueList ::= "(" ConstraintValue&Attributes { Comma ConstraintValue&Attributes }")"
355.ValueRange ::= "(" ValRange ")"
356.ValRange ::= (LowerRangeBound To UpperRangeBound)
357.LowerRangeBound ::= ConstraintExpression | Minus **INFINITY**
358.UpperRangeBound ::= ConstraintExpression | **INFINITY**
359.SuperSet ::= **SUPERSET** "(" ConstraintValue&Attributes ")"
360.SubSet ::= **SUBSET** "(" ConstraintValue&Attributes ")"
361.Permutation ::= **PERMUTATION** ValueList
362.ValueAttributes ::= [ValueLength] [**IF_PRESENT**] [ASN1_Encoding]
363.ASN1_Encoding ::= **ENC** PDU_FieldEncodingCall
364.ValueLength ::= SingleValueLength | RangeValueLength
365.SingleValueLength ::= "[" ValueBound "]"
366.ValueBound ::= Number | TS_ParIdentifier | TS_ConstIdentifier | FormalParIdentifier

367.RangeValueLength ::= "[" LowerValueBound To UpperValueBound "]"
368.LowerValueBound ::= ValueBound
369.UpperValueBound ::= ValueBound | **INFINITY**

ASN.1 PDU Constraint Declarations

370.ASN1_PDU_Constraints ::= **$ASN1_PDU_Constraints** {ASN1_PDU_Constraint}+ **$End_ASN1_PDU_Constraints**
371.ASN1_PDU_Constraint ::= **$Begin_ASN1_PDU_Constraint** ConsId PDU_Id DerivPath [EncRuleId] [EncVariationId] [Comment] [ASN1_ConsValue] [Comment] **$End_ASN1_PDU_Constraint**
372.ASN1_ConsValue ::= **$ASN1_ConsValue** ConstraintValue&AttributesOrReplace **$End_ASN1_ConsValue**
373.ConstraintValue&AttributesOrReplace ::= ConstraintValue&Attributes | Replacement {Comma Replacement}
374.Replacement ::= (**REPLACE** ReferenceList **BY** ConstraintValue&Attributes) | (**OMIT** ReferenceList)
375.ReferenceList ::= (ArrayRef | ComponentIdentifier | ComponentPosition) {ComponentReference}

CM Constraint Declarations

376.CM_Constraints ::= **$CM_Constraints** [TTCN_CM_Constraints] [ASN1_CM_Constraints] **$End_CM_Constraints**

Tabular CM Constraint Declaration

377.TTCN_CM_Constraints ::= **$TTCN_CM_Constraints** {TTCN_CM_Constraint}+ **$End_TTCN_CM_Constraints**
378.TTCN_CM_Constraint ::= **$Begin_TTCN_CM_Constraint** ConsId CM_Id DerivPath [Comment] [CM_ParValues] [Comment] **$End_TTCN_CM_Constraint**
379.CM_ParValues ::= **$CM_ParValues** {CM_ParValue} **$End_CM_ParValues**
380.CM_ParValue ::= **$CM_ParValue** CM_ParId ConsValue [Comment] **$End_CM_ParValue**

ASN.1 CM Constraint Declaration

381.ASN1_CM_Constraints ::= **$ASN1_CM_Constraints** {ASN1_CM_Constraint}+ **$End_ASN1_CM_Constraints**
382.ASN1_CM_Constraint ::= **$Begin_ASN1_CM_Constraint** ConsId CM_Id DerivPath [Comment] [ASN1_ConsValue] [Comment] **$End_ASN1_CM_Constraint**

The Dynamic Part

383.DynamicPart ::= **$DynamicPart** [TestCases] [TestStepLibrary] [DefaultsLibrary] **$End_DynamicPart**

Test Cases

384.TestCases ::= **$TestCases** {TestGroup | TestCase | *CompactTestGroup*}+ **$End_TestCases**
385.TestGroup ::= **$TestGroup** TestGroupId *[SelExprId] [Objective]* {TestGroup | TestCase}+ **$End_TestGroup**
386.TestGroupId ::= **$TestGroupId** TestGroupIdentifier
387.TestGroupIdentifier ::= Identifier
388.TestCase ::= **$Begin_TestCase** TestCaseId TestGroupRef TestPurpose [Configuration] DefaultsRef [Comment] *[SelExprId] [Description]* BehaviourDescription [Comment] **$End_TestCase**
389.TestCaseId ::= **$TestCaseId** TestCaseIdentifier
390.TestCaseIdentifier ::= Identifier
391.TestGroupRef ::= **$TestGroupRef** TestGroupReference
392.TestGroupReference ::= [SuiteIdentifier "/"] {TestGroupIdentifier "/"}
393.TestPurpose ::= **$TestPurpose** BoundedFreeText
394.Configuration ::= **$Configuration** TCompConfigIdentifier
395.DefaultsRef ::= **$DefaultsRef** [DefaultRefList]
396.DefaultRefList ::= DefaultReference {Comma DefaultReference}
397.DefaultReference ::= DefaultIdentifier [ActualParList]
397a.CompactTestGroup ::= **$Begin_CompactTestGroup** TestGroupId DefaultsRef *[SelExprId] [Objective]* {CompactTestCase} **$End_CompactTestGroup**
397b.CompactTestCase ::= **$CompactTestCase** TestCaseId TestPurpose TestStepAttachment [Comment] *[SelExprId] [Description]* **$End_CompactTestCase**
397c.TestStepAttachment ::= **$TestStepAttachment** Attach

Test Step Library

398.TestStepLibrary ::= **$TestStepLibrary** {TestStepGroup | TestStep}+ **$End_TestStepLibrary**
399.TestStepGroup ::= **$TestStepGroup** TestStepGroupId {TestStepGroup | TestStep}+ **$End_TestStepGroup**
400.TestStepGroupId ::= **$TestStepGroupId** TestStepGroupIdentifier
401.TestStepGroupIdentifier ::= Identifier
402.TestStep ::= **$Begin_TestStep** TestStepId TestStepRef Objective DefaultsRef [Comment] *[Description]* BehaviourDescription [Comment] **$End_TestStep**
403.TestStepId ::= **$TestStepId** TestStepId&ParList
404.TestStepId&ParList ::= TestStepIdentifier [FormalParList]
405.TestStepIdentifier ::= Identifier
406.TestStepRef ::= **$TestStepRef** TestStepGroupReference
407.TestStepGroupReference ::= [SuiteIdentifier "/"] {TestStepGroupIdentifier "/"}
408.Objective ::= **$Objective** BoundedFreeText

Default Library

409.DefaultsLibrary ::= **$DefaultsLibrary** {DefaultGroup | Default}+
$End_DefaultsLibrary
410.DefaultGroup ::= **$DefaultGroup** DefaultGroupId {DefaultGroup | Default}+
$End_DefaultGroup
411.DefaultGroupId ::= **$DefaultGroupId** DefaultGroupIdentifier
412.Default ::= **$Begin_Default** DefaultId DefaultRef Objective [Comment] *[Description]* BehaviourDescription [Comment] **$End_Default**
413.DefaultRef ::= **$DefaultRef** DefaultGroupReference
414.DefaultId ::= **$DefaultId** DefaultId&ParList
415.DefaultId&ParList ::= DefaultIdentifier [FormalParList]
416.DefaultIdentifier ::= Identifier
417.DefaultGroupReference ::= [SuiteIdentifier "/"] {DefaultGroupIdentifier "/"}
418.DefaultGroupIdentifier ::= Identifier

Behaviour descriptions

419.BehaviourDescription ::= **$BehaviourDescription** RootTree {LocalTree}
$End_BehaviourDescription
420.RootTree ::= {BehaviourLine}+
421.LocalTree ::= Header {BehaviourLine}+
422.Header ::= **$Header** TreeHeader
423.TreeHeader ::= TreeIdentifier [FormalParList]
424.TreeIdentifier ::= Identifier
425.FormalParList ::= "(" FormalPar&Type {SemiColon FormalPar&Type} ")"
426.FormalPar&Type ::= FormalParIdentifier {Comma FormalParIdentifier} Colon FormalParType
427.FormalParIdentifier ::= Identifier
428.FormalParType ::= Type | PCO_TypeIdentifier | **PDU** | **CP** | **TIMER**

Behaviour lines

429.BehaviourLine ::= **$BehaviourLine** LabelId Line Cref VerdictId [Comment] **$End_BehaviourLine**
430.Line ::= **$Line** Indentation StatementLine
431.Indentation ::= "[" Number "]"
432.LabelId ::= **$LabelId** [Label]
433.Label ::= Identifier
434.Cref ::= **$Cref** [ConstraintReference]
435.ConstraintReference ::= ConsRef | FormalParIdentifier | AnyValue
436.ConsRef ::= ConstraintIdentifier [ActualCrefParList]
437.ActualCrefParList ::= "(" ActualCrefPar {Comma ActualCrefPar} ")"
438.ActualCrefPar ::= Value
439.VerdictId ::= **$VerdictId** [Verdict]
440.Verdict ::= Pass | Fail | Inconclusive | Result
441.Pass ::= **PASS** | **P** | "(" **PASS** ")" | "(" **P** ")"
442.Fail ::= **FAIL** | **F** | "(" **FAIL** ")" | "(" **F** ")"
443.Inconclusive ::= **INCONC** | **I** | "(" **INCONC** ")" | "(" **I** ")"
444.Result ::= **R**

TTCN statements

445.StatementLine ::= (Event [Qualifier] [AssignmentList] [TimerOps]) | (Qualifier [AssignmentList] [TimerOps]) | (AssignmentList [TimerOps]) | TimerOps | Construct | ImplicitSend
446.Event ::= Send | Receive | Otherwise | Timeout | Done
447.Qualifier ::= "[" Expression "]"
448.Send ::= [PCO_Identifier | CP_Identifier | FormalParIdentifier] "!" (ASP_Identifier | PDU_Identifier | CM_Identifier)
449.ImplicitSend ::= "<" **IUT** "!" (ASP_Identifier | PDU_Identifier) ">"
450.Receive ::= [PCO_Identifier | CP_Identifier | FormalParIdentifier] "?" (ASP_Identifier | PDU_Identifier | CM_Identifier)
451.Otherwise ::= [PCO_Identifier | FormalParIdentifier] "?" **OTHERWISE**
452.Timeout ::= "?" **TIMEOUT** [TimerIdentifier | FormalParIdentifier]
453.Done ::= "?" **DONE** "(" [TCompIdList] ")"
454.TCompIdList ::= TCompIdentifier {Comma TCompIdentifier}
455.Construct ::= GoTo | Attach | Repeat | Return | Activate | Create
456.Activate ::= **ACTIVATE** "(" [DefaultRefList] ")"
457.Return ::= **RETURN**
458.Create ::= **CREATE** "(" CreateList ")"
459.CreateList ::= CreateTComp {Comma CreateTComp}
460.CreateTComp ::= TCompIdentifier Colon TreeReference [ActualParList]
461.GoTo ::= ("->" | **GOTO**) Label

462.Attach ::= "+" TreeReference [ActualParList]
463.Repeat ::= **REPEAT** TreeReference [ActualParList] **UNTIL** Qualifier
464.TreeReference ::= TestStepIdentifier | TreeIdentifier
465.ActualParList ::= "(" ActualPar {Comma ActualPar} ")"
466.ActualPar ::= Value | PCO_Identifier | CP_Identifier | TimerIdentifier

Expressions

467.AssignmentList ::= "(" Assignment {Comma Assignment} ")"
468.Assignment ::= DataObjectReference ":=" Expression
469.Expression ::= SimpleExpression [RelOp SimpleExpression]
470.SimpleExpression ::= Term {AddOp Term}
471.Term ::= Factor {MultiplyOp Factor}
472.Factor ::= [UnaryOp] Primary
473.Primary ::= Value | DataObjectReference | OpCall | SelectExprIdentifier | "(" Expression ")"
474.DataObjectReference ::= DataObjectIdentifier {ComponentReference}
475.DataObjectIdentifier ::= TS_ParIdentifier | TS_ConstIdentifier | TS_VarIdentifier | TC_VarIdentifier | FormalParIdentifier | ASP_Identifier | PDU_Identifier | CM_Identifier
476.ComponentReference ::= RecordRef | ArrayRef | BitRef
477.RecordRef ::= Dot (ComponentIdentifier | PDU_Identifier | StructIdentifier | ComponentPosition)
478.ComponentIdentifier ::= ASP_ParIdentifier | PDU_FieldIdentifier | CM_ParIdentifier | ElemIdentifier | ASN1_Identifier
479.ASN1_Identifier ::= Identifier
480.ComponentPosition ::= ("Number")"
481.ArrayRef ::= Dot "[" ComponentNumber "]"
482.ComponentNumber ::= Expression
483.BitRef ::= Dot (BitIdentifier | "[" BitNumber "]")
484.BitIdentifier ::= Identifier
485.BitNumber ::= Expression
486.OpCall ::= OpIdentifier (ActualParList | "(" ")")
487.OpIdentifier ::= TS_OpIdentifier | PredefinedOpIdentifier
488.PredefinedOpIdentifier ::= BIT_TO_INT | HEX_TO_INT | INT_TO_BIT | INT_TO_HEX | IS_CHOSEN | IS_PRESENT | LENGTH_OF | NUMBER_OF_ELEMENT
489.AddOp ::= "+" | "-" | **OR**
490.MultiplyOp ::= "*" | "/" | **MOD** | **AND**
491.UnaryOp ::= "+" | "-" | **NOT**
492.RelOp ::= "=" | "<" | ">" | "<>" | ">=" | "<="

Timer operations

493.TimerOps ::= TimerOp {Comma TimerOp}
494.TimerOp ::= StartTimer | CancelTimer | ReadTimer
495.StartTimer ::= **START** TimerIdentifier ["(" TimerValue ")"]
496.CancelTimer ::= **CANCEL** [TimerIdentifier]
497.TimerValue ::= Expression
498.ReadTimer ::= **READTIMER** TimerIdentifier "(" DataObjectReference")"

Types

499.TypeOrPDU ::= Type | PDU
500.Type ::= PredefinedType | ReferenceType

Predefined types

501.PredefinedType ::= **INTEGER** | **BOOLEAN** | **BITSTRING** | **HEXSTRING** | **OCTETSTRING** | **R_Type** | CharacterString
502.CharacterString ::= **NumericString** | **PrintableString** | **TeletexString** | **VideotexString** | **VisibleString** | **IA5String** | **GraphicString** | **GeneralString** | **T61String** | **ISO646String**

Referenced types

503.ReferenceType ::= TS_TypeIdentifier | ASP_Identifier | PDU_Identifier | CM_Identifier
504.TS_TypeIdentifier ::= SimpleTypeIdentifier | StructIdentifier | ASN1_TypeIdentifier

Values

505.Value ::= LiteralValue | ASN1_Value [ASN1_Encoding]
506.LiteralValue ::= Number | BooleanValue | Bstring | Hstring | Ostring | Cstring | R_Value
507.Number ::= (NonZeroNum {Num}) | **0**
508.NonZeroNum ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
509.Num ::= **0** | NonZeroNum
510.BooleanValue ::= **TRUE** | **FALSE**
511.Bstring ::= "'" {Bin | Wildcard} "'" **B**
512.Bin ::= **0** | **1**
513.Hstring ::= "'" {Hex | Wildcard} "'" **H**
514.Hex ::= Num **| A | B | C | D | E | F**
515.Ostring ::= "'" {Oct | Wildcard} "'" **O**
516.Oct ::= Hex Hex
517.Cstring ::= """ {Char | Wildcard | "\"} """
518.Char ::= */* REFERENCE - A character defined by the relevant character string type */*

519.Wildcard ::= AnyOne | AnyOrNone
520.AnyOne ::= "?"
521.AnyOrNone ::= "*"
522.R_Value ::= **pass** | **fail** | **inconc** | **none**
523.Identifier ::= Alpha{AlphaNum | Underscore}
524.Alpha ::= UpperAlpha | LowerAlpha
525.AlphaNum ::= Alpha | Num
526.UpperAlpha ::= **A | B| C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z**
527.LowerAlpha ::= **a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z**
528.ExtendedAlphaNum ::= */* REFERENCE - A character from any character set defined in ISO/IEC 10646 */*
529.BoundedFreeText ::= *"/*"* FreeText *"*/"*
530.FreeText ::= {ExtendedAlphaNum}

Miscellaneous productions

531.Comma ::= ","
532.Dot ::= "."
533.Dash ::= "-"
534.Minus ::= "-"
535.SemiColon ::= ";"
536.Colon ::= ":"
537.Underscore ::= "_"

The ASN1 Syntax Productions in BNF

1. moduleidentifier ::= BoundedFreeText
2. typereference ::= Identifier
3. valuereference ::= Identifier
4. TypeAssignment ::= typereference "::=" ASN1_main_Type
5. ASN1_main_Type ::= BuiltinType | DefinedType | SubType
6. BuiltinType ::= BooleanType | IntegerType | BitStringType | OctetStringType | NullType | SequenceType | SequenceOfType | SetType | SetOfType | ChoiceType | SelectionType | TaggedType | AnyType | ObjectIdentifierType | CharacterStringType | UsefulType | EnumeratedType | RealType
7. NamedType ::= Identifier ASN1_main_Type | ASN1_main_Type | SelectionType
8. ASN1_Value ::= BuiltinValue | DefinedValue
9. BuiltinValue ::= BooleanValue | IntegerValue | BitStringValue | OctetStringValue | NullValue | SequenceValue | SequenceOfValue | SetValue | SetOfValue | ChoiceValue | SelectionValue | TaggedValue | ASN1_AnyValue | ObjectIdentifierValue | CharacterStringValue | EnumeratedValue | RealValue
10. NamedValue ::= Identifier ASN1_Value | ASN1_Value
11. BooleanType ::= BOOLEAN
12. IntegerType ::= INTEGER | INTEGER "{" NamedNumberList "}"
13. NamedNumberList ::= NamedNumber | NamedNumberList Comma NamedNumber
14. NamedNumber ::= Identifier "(" Minus Number ")" | Identifier "(" Number ")" | Identifier "(" DefinedValue ")"
15. IntegerValue ::= Minus Number | Number | Identifier
16. EnumeratedType ::= ENUMERATED "{" Enumeration "}"
17. Enumeration ::= NamedNumber | Enumeration Comma NamedNumber
18. EnumeratedValue ::= Identifier
19. RealType ::= REAL
20. RealValue ::= NumericalRealValue | SpecialRealValue
21. NumericalRealValue ::= "{" Mantissa Comma Base Comma Exponent "}"
22. Mantissa ::= Minus Number | Number
23. Base ::= Two | Ten
24. Exponent ::= Minus Number | Number
25. SpecialRealValue ::= PLUS_INFINITY | MINUS_INFINITY
26. BitStringType ::= BIT STRING | BIT STRING "{" NamedBitList "}"
27. NamedBitList ::= NamedBit | NamedBitList Comma NamedBitList
28. NamedBit ::= Identifier "(" Number ")" | Identifier "(" DefinedValue ")"
29. BitStringValue ::= Bstring | Hstring | "{" IdentifierList "}" | "{" "}"
30. IdentifierList ::= Identifier | IdentifierList Comma Identifier
31. OctetStringType ::= OCTET STRING
32. OctetStringValue ::= Bstring | Hstring

33. NullType ::= NULL
34. NullValue ::= NULL
35. SequenceType ::= SEQUENCE "{" ElementTypeList "}"
36. ElementTypeList ::= | ElementType | ElementTypeList Comma ElementType
37. ElementType ::= NamedType | NamedType OPTIONAL | NamedType DEFAULT ASN1_Value | COMPONENTS OF ASN1_main_Type
38. SequenceValue ::= "{" ElementValueList "}" | "{" "}"
39. ElementValueList ::= NamedValue | ElementValueList Comma NamedValue
40. SequenceOfType ::= SEQUENCE OF ASN1_main_Type | SEQUENCE
41. SequenceOfValue ::= LBRACE ASN1_ValueList RBRACE | LBRACE RBRACE
42. ASN1_ValueList ::= ASN1_Value | ASN1_ValueList Comma ASN1_Value
43. SetType ::= SET "{" ElementTypeList "}"
44. SetValue ::= "{" ElementValueList "}" | "{" "}"
45. SetOfType ::= SET OF ASN1_main_Type | SET
46. SetOfValue ::= "{" ASN1_ValueList "}" | "{" "}"
47. ChoiceType ::= CHOICE "{" AlternativeTypeList "}"
48. AlternativeTypeList ::= NamedType | AlternativeTypeList Comma NamedType
49. ChoiceValue::=Identifier : Value
50. SelectionType ::= Identifier "<" ASN1_main_Type
51. SelectionValue ::= NamedValue
52. TaggedType ::= Tag ASN1_main_Type | Tag IMPLICIT ASN1_main_Type | Tag EXPLICIT ASN1_main_Type
53. Tag ::= "(" Class ClassNumber ")"
54. Class ::= UNIVERSAL | APPLICATION | PRIVATE | empty
55. ClassNumber ::= Number | DefinedValue
56. TaggedValue ::= ASN1_Value
57. AnyType ::= ANY | ANY DEFINED_ BY Identifier
58. ASN1_AnyValue ::= ASN1_main_Type Colon ASN1_Value
59. ObjectIdentifierType ::= OBJECT IDENTIFIER
60. ObjectIdentifierValue ::= "{" ObjIdComponentList "}" | "{" DefinedValue ObjIdComponentList "}"
61. ObjIdComponentList ::= ObjComponent | ObjIdComponentList ObjComponent
62. ObjComponent ::= NameForm | NumberForm | NameAndNumberForm
63. NameForm ::= Identifier
64. NumberForm ::= Number | DefinedValue
65. NameAndNumberForm ::= Identifier "{" NumberForm "}"
66. CharacterStringType ::= Identifier
67. CharacterStringValue ::= Cstring
68. UsefulType ::= Identifier
69. SubType ::= ParentType SubtypeSpec | SET SizeConstraint OF ASN1_main_Type | SEQUENCE SizeConstraint OF ASN1_main_Type

70. ParentType ::= ASN1_main_Type
71. SubtypeSpec ::= "(" SubtypeValueSetList ")"
72. SubtypeValueSetList ::= SubtypeValueSet | SubtypeValueSetList '|' SubtypeValueSet
73. SubtypeValueSet ::= ASN1_Value | ContainedSubtype | ASN1_ValueRange | PermittedAlphabet | SizeConstraint | InnerTypeConstraints
74. ContainedSubtype ::= INCLUDES ASN1_main_Type
75. ASN1_ValueRange ::= LowerEndpoint DotDot UpperEndpoint
76. LowerEndpoint ::= LowerEndValue | LowerEndValue "<"
77. LowerEndValue ::= ASN1_Value | MIN
78. UpperEndpoint ::= UpperEndValue | "<" UpperEndValue
79. UpperEndValue ::= ASN1_Value | MAX
80. SizeConstraint ::= SIZE SubtypeSpec
81. PermittedAlphabet ::= FROM SubtypeSpec
82. InnerTypeConstraints ::= WITH COMPONENT SingleTypeConstraint | WITH COMPONENTS MultipleTypeConstraints
83. SingleTypeConstraint ::= SubtypeSpec
84. MultipleTypeConstraints ::= FullSpecification | PartialSpecification
85. FullSpecification ::= "{" TypeConstraints "}"
86. PartialSpecification ::= "{" DotDotDot Comma TypeConstraints "}"
87. TypeConstraints ::= NamedConstraint | NamedConstraint Comma TypeConstraints
88. NamedConstraint ::= Identifier Constraint | Constraint
89. Constraint ::= ValueConstraint PresenceConstraint
90. ValueConstraint ::= SubtypeSpec | empty
91. PresenceConstraint ::= PRESENT | ABSENT | OPTIONAL | empty

TTCN Static Semantics

The following are those static semantics from the TTCN and ASN.1 standards which are checked by the Analyzer tool in the TTCN Suite.

Test Suite

SuiteId shall be the same as the SuiteId declared in Test Suite Structure table (Suite Structure).

Test Case Index

Test Cases shall be listed in the order that they exist in the dynamic part.

Test Step Index

TestStepId shall not include a formal parameter list.

Test Steps shall be listed in the order that they exist in the dynamic part.

Default Index

DefaultId shall not include a formal parameter list.

Defaults shall be listed in the order that they exist in the dynamic part.

Test Suite Type Definitions

Wherever types are referenced within Test Suite Type Definitions those references shall not be recursive (neither directly or indirectly)

Simple Type Definitions

The base type[1] shall be a Predefined Type or a Simple Type.

The set of values defined by a restriction must be a true subset of the values of the base type.

In specification of a particular length range, only non-negative INTEGER literals or the keyword INFINITY for the upper bound shall be used.

1. By Base Type we refer to the particular type in the Type production

If Minus is used in SimpleValueList then LiteralValue shall be a number.

The restriction type shall be a list of distinguished values of the base type.

Where a range is used in a type definition either as a value range or as a length range (for strings) it shall be stated with the lower of the two values on the left.

An integer range shall be used only with a base type of INTEGER or a type derived from INTEGER.

Where a value list is used, the values shall be of the base type and shall be a true subset of the values defined by the base type.

LengthRestriction shall be provided only when the base type is a string type (i.e., BITSTRING, HEXSTRING, OCTETSTRING or CharacterString) or derived from a string type

The LiteralValues shall be of the base type

In RangeTypeLength:
LowerTypeBound shall be a non-negative number
LowerTypeBound shall be less than UpperTypeBound

In IntegerRange:
LowerTypeBound shall be less than UpperTypeBound

Structured Type Definitions

Where elements may be of a type of arbitrarily complex structure; there shall be no recursive references.

The elements of Structured Type definitions are considered to be optional, i.e., in instances of these types whole elements may not be present.

A structure element Type shall be a PredefinedType, TS_TypeIdentifier, PDU_Identifier, or PDU.

If a Structured Type is used as a macro expansion, then the names of the elements within the Structured Type shall be unique within each ASP or PDU where it will be expanded.

The optional element length restriction can be used in order to give the minimum and maximum length of an element of a string type.

The set of values defined by LengthAttribute shall be a subset of the values of the base type.

ASN1 Type Definition

Types referred to from the type definition shall be defined in other ASN.1 type definition tables, be defined by reference in the ASN.1 type reference table or be defined locally in the same table, following the first type definition. Locally defined types shall not be used in other parts of the test suite.

ASN.1 type definitions used within TTCN shall not use external type references as defined in ISO/IEC 8824.

When ASN1 is used in a TTCN test suite, ASN1 identifiers from the following list shall be unique throughout the test suite:
a) identifiers occurring in an ASN1ENUMERATED type as distinguished values
b) identifiers occurring in a NamedNumberList of an ASN1 INTEGER type

Each terminal type reference used within the Type production shall be one of the following: ASN1_LocalType type reference, TS_TypeIdentifier or PDU_Identifier.

Test Suite Operation Definition

Only predefined types and data types as defined in the Test Suite Type definitions, ASP type definitions or PDU type definitions may be used as types for formal parameters. PCO types shall not be used as formal parameter types.

When a Test Suite Operation is invoked

1. the number of the actual parameters shall be the same as the number of the formal parameters; and

2. each actual parameter shall evaluate to an element of its corresponding formal type of parameter's.

Type in ResultType shall be a Predefined Type, TS_TypeIdentifier, PDU_Identifier or ASP_Identifier.

Test Suite Parameters Declarations

The type shall be a Predefined Type, TS_TypeIdentifier, PDU_Identifier or ASP_Identifier.

Test Case Selection Expression Definition

The expression shall use only literal values, Test Suite Parameters, Test Suite Constants and other selection expression identifiers in its terms.

The expression shall evaluate to a BOOLEAN value.

Expression shall not recursively refer (neither directly nor indirectly) to the SelExprIdentifier being defined by that Expression.

Test Suite Constant Declarations

The type shall be a predefined type, an ASN.1 type, a Test Suite Type or a PDU type.

The terms in the value expression shall not contain: Test Suite Variables or Test Case Variables.

The value shall evaluate to an element of the type indicated in the type column.

Test Suite Variable Declarations

Type shall be a Predefined Type, TS_TypeIdentifier, PDU_Identifier or ASP_Identifier.

The terms in the value expression shall not contain: Test Suite Variables or Test Case Variables.

Specifying an initial value is optional.

If an unbound Test Suite Variable is used in the right-hand side of an assignment, then it is a test case error.

The value shall evaluate to an element of the type indicated in the type column.

Test Case Variable Declarations

Type shall be a Predefined Type, TS_TypeIdentifier, PDU_Identifier or ASP_Identifier.

The terms in the value expression shall not contain: Test Suite Variables or Test Case Variables.

Specifying an initial value is optional.

The value shall evaluate to an element of the type indicated in the type column.

PCO Declarations

It is possible to define a PCO to correspond to a *set* of SAPs.

Timer Declarations

The terms in the value expression shall not contain: Test Suite Variables or Test Case Variables.

The timer duration shall evaluate to an unsigned positive INTEGER value.

ASP Type Definition

ASP type definitions may include ASN1 type definitions, if appropriate.

If only a single PCO is defined within a test suite then PCO_TypeIdentifier is optional. The PCO type shall be one of the PCO types used in the PCO Type Declaration proforma.

The macro symbol shall be used only with Structured Types defined in the Structured Types definitions.

Parameters may be of a type of arbitrarily complex structure, including being specified as a Test Suite Type (either predefined, Simple Type, Structured Type or ASN.1 type).

If a parameter is to be structured as a PDU, then its type may be stated either:

- as a PDU identifier to indicate that in the constraint for the ASP this parameter may be chained to a PDU constraint of a specific PDU type.
- as **PDU** to indicate that in the constraint for the ASP this parameter may be chained to a PDU constraint of any PDU type.

The boundaries shall be specified in terms of (non-negative) INTEGER literals, Test Suite Parameters, Test Suite Constants or the keyword INFINITY.

The parameters of ASP type definitions are considered to be optional, i.e., in instances of these types whole parameters may not be present.

The names of ASP parameters shall be unique within the ASP in which they are declared.

The optional attribute is Length.

In ASPs that are sent from the tester, values for ASP parameters that are defined in the Constraints Part shall correspond to the parameter or field definition. This means:
a) the value shall be of the type specified for that ASP parameter; and
b) In the case of substructured ASPs, either using Structured Types or ASN.1, the above rules apply to the fields of the substructure(s) recursively.

ASN1 ASP Type Definitions

The PCO type shall be one of the PCO types used in the PCO declaration proforma.

If only a single PCO is defined within a test suite, specifying the PCO type in an ASP type definition is optional.

Types referred to from the ASP definition shall be defined in other ASN.1 type definition tables, be defined by reference in the ASN.1 type reference table or be defined locally in the same table, following the first type definition.

Locally defined types shall not be used in other parts of the test suite.

ASN1 ASP Type Definition By Reference

The PCO type shall be one of the PCO types used in the PCO declaration proforma.

If only a single PCO is defined within a test suite, specifying the PCO type in an ASP type definition is optional.

PDU Type Definition

The PCO type shall be one of the PCO types used in the PCO declaration proforma.

The macro symbol shall be used only with Structured Types defined in the Structured Types definitions.

Fields may be of a type of arbitrarily complex structure, including being specified as a Test Suite Type (either predefined, Simple Type, Structured Type or ASN.1 type).

If a field is to be structured as a PDU, then its type may be stated either

- as a PDU identifier to indicate that in the constraint for the PDU this field may be chained to a PDU constraint of a specific PDU type; or
- as PDU to indicate that in the constraint for the PDU this field may be chained to a PDU constraint of any PDU type.

The boundaries shall be specified in terms of non-negative INTEGER literals, Test Suite Parameters, Test Suite Constants or the keyword INFINITY.

The fields of PDU type definitions are considered to be optional, i.e., in instances of these types whole fields may not be present.

The names of PDU fields shall be unique within the PDU in which they are declared

The optional attribute is Length;

In PDUs that are sent from the tester, values for PDU fields that are defined in the Constraints Part shall correspond to the field definition. This means

1. that the value shall be of the type specified for that PDU field; and
2. in the case of substructured ASPs and/or PDUs, either using Structured Types or ASN.1, the above rules apply to the fields of the substructure(s) recursively.

The set of values defined by LengthAttribute shall be a true subset of the values of the base type.

LengthAttribute shall be provided only when the base type is a string type (i.e. BITSTRING, HEXSTRING, OCTETSTRING or CharacterString) or derived from a string type.

ASN1 PDU Type Definition

The PCO type shall be one of the PCO types used in the PCO declaration proforma.

If only a single PCO is defined within a test suite, specifying the PCO type in an PDU type definition is optional.

Types referred to from the ASP definition shall be defined in other ASN.1 type definition tables, be defined by reference in the ASN.1 type reference table or be defined locally in the same table, following the first type definition.

Locally defined types shall not be used in other parts of the test suite.

ASN1 PDU Type Definition By Reference

The PCO type shall be one of the PCO types used in the PCO declaration proforma.

If only a single PCO is defined within a test suite, specifying the PCO type in an PDU type definition is optional.

String Length Specifications

TTCN permits the specification of length restrictions on string types (i.e., BITSTRING, HEXSTRING, OCTETSTRING and all CharacterString types) in the following instances:

1. when declaring Test Suite Types as a type restriction;
2. when declaring simple ASP parameters, PDU fields and elements of Structured Types as an attribute of the parameter, field or element type;
   and
3. when defining ASP/PDU or Structured Type constraints as an attribute of the constraint value. In the context of constraints, length restrictions can also be specified on values of type SEQUENCE OF or SET OF, thus limiting the number of their elements.

Alias Definitions

An Alias shall be used only to replace an ASP identifier or a PDU identifier within a single TTCN statement in a behaviour tree. It shall be used only in a behaviour description column.

Structured Type Constraint Declarations

If an ASP or PDU definition refers to a Structured Type as a substructure of a parameter or field (i.e., with a parameter name or a field name specified for it) then the corresponding constraint shall have the same parameter or field name in the corresponding position in the parameter name or field name column of the constraint and the value shall be a reference to a constraint for that parameter or field (i.e., for that substructure in accordance with the definition of the Structured Type).

ASP Constraint Declarations

If the ASP definition refers to a Structured Type by macro expansion (i.e., with <- in place of the ASP field name) then in a corresponding constraint either

- the individual elements from the Structured Type shall be included directly within the constraints.

- the macro symbol (<-) shall be placed in the corresponding position in the ASP field name column of the constraint and the value shall be a reference to a constraint for the Structured Type referenced from the ASP definition.

PDU Constraint Declarations

If the PDU definition refers to a Structured Type by macro expansion (i.e., with <- in place of the PDU field name) then in a corresponding constraint either:

- the individual elements from the Structured Type shall be included directly within the constraints;
  or
- the macro symbol (<-) shall be placed in the corresponding position in the PDU field name column of the constraint and the value shall be a reference to a constraint for the Structured Type referenced from the PDU definition.

Constraints Part

If an ASP and/or PDU is substructured, then the constraints for ASPs and/or PDUs of that type shall have the same tabular structure or a compatible ASN.1 structure (i.e., possibly with some groupings).

Structured Types expanded into an ASP or PDU definition by use of the macro symbol (<-) are not considered to be substructures. Constraints for such ASPs or PDUs shall either have a completely flat structure (i.e., the elements of an expanded structure are explicitly listed in the ASP or PDU constraint) or shall reference a corresponding structure constraint for macro expansion.

Whichever way the values are obtained, they shall correspond to the parameter or field entries in the ASP or PDU type definitions. This means

1. that the value shall be of the type specified for that parameter or field.
2. that the length shall satisfy any restriction associated with the type. (This will not be implemented.)

An expression in a constraint shall contain only literal values, Test Suite Parameters, Test Suite Constants, formal parameters and Test Suite Operations.

Neither Test Suite Variables nor Test Case Variables shall be used in constraints, unless passed as actual parameters.

Literal values, Test Suite Parameters, Test Suite Constants, Test Suite Variables, Test Case Variables and PDU or Test Suite Type constraints may be passed as actual parameters to a constraint in a constraints reference made from a behaviour description. The parameters shall not be of PCO type or ASP type.

In ASN.1 constraints, only ASP parameters and PDU fields declared as OPTIONAL may be omitted. These may be omitted either by using the Omit symbol or by simply leaving out the relevant ASP parameter or PDU field.

The constraint specification of an ASP and/or PDU shall have the same structure as that of the type definition of that ASP or PDU.

In tabular constraints, all ASP parameters and PDU fields are optional and therefore may be omitted using the Omit symbol, to indicate that the ASP parameter or PDU field is to be absent from the event sent.

An expression in a constraint shall contain only Values (including, for instance, ConstraintValue&Attributes), Test Suite Parameters, Test Suite Constraints, formal parameters, Component References and Test Suite Operations.

Matching Mechanisms

Complement: Each constraint value in the list shall be of the type declared for the ASP parameter or PDU field in which the complement mechanism is used.

ValueList: Each value in the ValueList shall be of the type declared for the ASP parameter or PDU field in which the ValueList mechanism is used.

Range: Ranges shall be used only on values of INTEGER type.

Range: A boundary value shall be either:

- INFINITY or -INFINITY
- a constraint expression that evaluates to a specific INTEGER value.

Range: The lower boundary shall be less than the upper boundary.

SuperSet: SuperSet is an operation for matching that shall be used on values of SET OF type. SuperSet shall be used only in ASN1 constraints.

SuperSet: The argument of SuperSet shall be of the type declared for the ASP parameter or PDU field in which the SuperSet mechanism is used.

SubSet: SubSet is an operation for matching that shall be used on values of SET OF type. SubSet shall be used only in ASN1 constraints.

SubSet: The argument of SubSet shall be of the type declared for the ASP parameter or PDU field in which the SubSet mechanism is used.

Permutation: Permutation an operation for matching that can be used only on values inside a value of SEQUENCE OF type. Permutation shall be used only in ASN1 constraints.

Permutation: Each element listed in Permutation shall be of the type declared inside the SEQUENCE OF type of the ASP parameter or PDU field.

Base Constraints and Modified Constraints

The name of the modified constraint shall be a unique identifier.

The name of the base constraint which is to be modified shall be indicated in the derivation path entry in the constraint header. This entry shall be left blank for a base constraint.

A modified constraint can itself be modified. In such a case the Derivation Path indicates the concatenation of the names of the base and previously modified constraints, separated by dots (.) A dot shall follow the last modified constraint name.

If a base constraint is defined to have a formal parameter list, the following rules apply to all modified constraints derived from that base

constraint, whether or not they are derived in one or several modification steps:

1. The modified constraint shall have the same parameter list as the base constraint. In particular, there shall be no parameters omitted from or added to this list
2. The formal parameter list shall follow the constraint name for every modified constraint
3. Parameterized ASP parameters or PDU in a base constraint fields shall not be modified or explicitly omitted in a modified constrain.

In tabular constraints Omit shall be denoted by dash (-). In ASN.1 constraints Omit is denoted by OMIT.

If the ASP or PDU definition refers to a parameter or field specified as being of metatype PDU then in a corresponding constraint the value for that parameter or field shall be specified as the name of a PDU constraint, or formal parameter.

The Behaviour Description

Statements in the first level of alternatives having no predecessor in the root or local tree belong to, shall have the indentation value of zero.

Statements having a predecessor shall have the indentation value of the predecessor plus one as their indentation value.

The parameters may provide PCOs, constraints, variables, or other such items for use within the tree.

Test Case root trees shall not be parameterized.

Formal parameters may be of PCO type, ASP type, PDU type, structure type or one of the other predefined or Test Suite Types.

TTCN Test Events

In the simplest form, an ASP identifier or PDU identifier follows the SEND symbol (!) for events to be initiated by the LT or UT, or a RECEIVE symbol (?) for events which it is possible for the LT or UT to accept.

If both a qualifier and an assignment are associated with the same event, then the qualifier shall appear first.

The tree header identifier used for local trees shall be unique within the dynamic behaviour description in which they appear, and shall not be the same as any identifier having a unique meaning throughout the test suite.

TTCN Expressions

The index notation is used to refer to elements (bits) of the ASN.1 BIT-STRING type. BITSTRING is assumed to be defined as SEQUENCE OF {BOOLEAN}. If certain bits of a BITSTRING are associated with an identifier (named bit) then either the dot notation or this identifier shall be used to refer to the bit.

Where a parameter, field or element is defined to be a true substructure of a type defined in a Structured Type table, a reference to the elements in the substructure shall consist of the reference to the parameter, field or element identifier followed by a dot and the identifier of the item within that substructure.

Where a structure is used as a macro expansion, the elements in the structure shall be referred to as if it was expanded into the structure referring to it.

If a parameter, field or element is defined to be of metatype PDU no reference shall be made to fields of that substructure.

The ATTACH Construct

Tree reference may be Test Step Identifiers or tree identifiers, where

1. A Test Step Identifier denotes the attachment of a Test Step that resides in the Test Step Library; the Test Step is referenced by its unique identifier
2. A tree identifier shall be the name of one of the trees in the current behaviour description; this is attachment of a local tree.

Constraints may be passed as parameters to Test Steps. If the constraint has a formal parameter list then the constraint shall be passed together with an actual parameter list.

When a parameterized tree is attached:

1. The number of the actual parameters shall be the same as the number of formal parameters
2. Each actual parameter shall evaluate to an element of its corresponding formal parameter type

Labels and the GOTO Construct

A GOTO to a label may be specified within a behaviour tree provided that the label is associated with the first of a set of alternatives, one of which is an ancestor node of the point from which the GOTO is to be made.

A GOTO shall be used only for jumps within one tree, i.e., within a Test Case root tree, a Test Step tree a Default tree or a local tree. As a consequence, each label used in a GOTO construct shall be found within the same tree in which the GOTO is used.

Labels used within a tree shall be unique within a tree.

The Constraints Reference

The actual parameter list shall fulfil the following:

1. the number of actual parameters shall be the same as the number of formal parameters;
   and
2. each actual parameter shall evaluate to an element of its corresponding formal type.

Verdicts

A predefined variable called R is available in each Test Case to store any Intermediate results. R can take the values *pass*, *fail*, *inconc* and *none*. These values are predefined identifiers and as such are case sensitive.

R shall not be used on the left-hand side of an assignment statement.

PASS or P, FAIL or F and INCONC or I are keywords that are used in the verdicts column only. The predefined identifiers pass, fail, inconc and none are values that represent the possible contents of the pre-

defined variable R. These predefined identifiers are to be used for testing the variable R in behaviour lines only.

A Verdict shall not occur in corresponding to entries in the behaviour tree which are any of the following: empty, an ATTACH construct, a GOTO construct, an IMPLICIT SEND or a RETURN.

Default References

Test Cases or Test Steps shall not be referred to as Defaults.

The actual parameter list shall fulfill the following:

1. The number of actual parameters shall be the same as the number of formal parameters.
2. Each actual parameter shall evaluate to an element of its corresponding formal type.
3. All variables appearing in the parameter list shall be bound when the constraint is invoked. (This won't be implemented.)

Formal Parameters

The formal parameter names which may optionally appear as part of the following shall be unique within that formal parameter list, and shall not be the same as any identifier having a unique meaning throughout the test suite.

A formal parameter name contained in the formal parameter list of a local tree shall header shall take precedence over a formal parameter name contained in the formal parameter list of the Test Step in which it is defined, within the scope of that local formal parameter list.

DataObjectReferences

A reference to a component of one of the following types: SEQUENCE, SET and CHOICE is constructed using a dot notation; i.e., appending a dot and the name (component identifier) of the desired component to the data object identifier; the component identifier shall be used if specified.

ASN.1 Static Semantics

IntegerType: Each identifier appearing in the NamedNumberList shall be different.

BitStringType: The DefinedValue shall be a reference to a value of type integer, or of a type derived from integer by tagging.

BitStringType: Each identifier appearing in the NamedNumberList shall be different.

BitStringValue: Each identifier in BitStringValue shall be the same as an identifier in the BitStringType sequence with which the value is associated.

SequenceType: The Type in the fourth alternative of the ElementType (COMPONENTS OF) shall be sequence type.

SequenceType: If OPTIONAL or DEFAULT are present, the corresponding value may be omitted from a value of the new type.

SequenceType: The identifiers in all NamedType sequence of the ElementTypeList shall be distinct.

SequenceType: The {} notation shall only be used if:
a) all ElementType sequences in the SequenceType are marked DEFAULT or OPTIONAL and all values are omitted
b) the type notation was SEQUENCE {}

SequenceType: There shall be one NamedValue for each NamedType in the SequenceType which is not marked OPTIONAL or DEFAULT, and the values shall be in the same order as the corresponding NamedType sequences.

SequenceOfType: Each Value sequence in the ValueList shall be the notation for a value of the Type specified in the SequenceOfType.

SequenceOfType: The {} notation is used when there are no component values in the sequence-of value.

SetType: The Type in the fourth alternative of the ElementType (COMPONENTS OF) shall be set type.

SetType: If OPTIONAL or DEFAULT are present, the corresponding value may be omitted from a value of the new type.

SetType: The identifiers in all NamedType sequence of the ElementTypeList shall be distinct.

SetType: The {} notation shall only be used if:
a) all ElementType sequences in the SetType are marked DEFAULT or OPTIONAL and all values are omitted
b) the type notation was SET {}

SetOfType: Each Value sequence in the ValueList shall be the notation for a value of the Type specified in the SetOfType.

SetOfType: The {} notation is used when there are no component values in the set-of value.

ChoiceType: The identifiers in all NamedType sequences of the AlternativeTypeList shall be distinct.

ChoiceType: If the NamedValue contains an identifier (in our case it contains always), it shall be a notation for a value of that type in the AlternativeTypeList that is named by the same identifier.

SelectionType: Type is a notation referencing the ChoiceType, and identifier is the identifier in the NamedType.

Subtype: When the SubtypeSpec notation follows the SelectionType notation, the parent type is the SelectionType, not the Type in the SelectionType notation.

Subtype: When the SubtypeSpec notation follows a set-of or sequence-of type notation, it applies to the Type in the set-of or sequence-of notation, not to the set-of or sequence-of type.

Chapter

39 *Using Diagram Editors*

This chapter describes the diagram editors you use for creating, drawing and printing OM (Object Models), SC (State Charts), DP (Deployment), MSC (Message Sequence Charts) and HMSC (High-level Message Sequence Charts) diagrams. Another name for HMSC is "road map".

The MSC Editor is also available in a flavor called *MSC Tracer* where the user is not allowed to add or remove symbols. This is only possible by an API which enables custom applications to interactively produce an MSC which reflects user defined events. Some features in the MSC Editor are not available in the MSC Tracer. For further information on this API, see the header file `<product install dir>/include/msctrace/msctrace.h`.

This chapter contains information about the functionality, menus, dialogs and windows of the diagram editors. For a guide to how to create and edit MSCs, see chapter 41, *Editing MSC Diagrams*.

Note: SDL Editor

The SDL editor is not covered in this chapter. For information regarding the SDL editor please see chapter 43, *Using the SDL Editor, in the User´s Manual*.

General

The editor described in this chapter is a combined OM, SC, DP, MSC and HMSC Editor. This editor is thus capable of showing five different types of diagrams, namely the Object Model (OM) diagrams, State Chart (SC) diagrams, Deployment (DP) diagrams, Message Sequence Chart (MSC) diagrams, and High-level MSC (HMSC) diagrams.

This combined editor is called **the editor** throughout this chapter. If only a specific editor is applied, that editor will be explicitly mentioned.

OM, SC, DP and HMSC diagrams are handled by one editor window, whereas MSC diagrams are handled by a separate editor window. This means that two editor windows are needed to handle all five diagram types.

The editor handling SDL diagrams is described in chapter 43, *Using the SDL Editor*.

How to edit MSC diagrams is described more closely in chapter 41, *Editing MSC Diagrams*.

Diagrams and Pages

The handling of diagrams and pages is common to the OM, SC, DP, MSC and HMSC diagrams. The editor can handle any number of diagrams.

The pages that the editor displays are always contained within a diagram. A diagram can contain any number of pages, but, must contain **at least one page**. The MSC and the DP Editor are exceptions as they always contain only **one** page. This page is enlarged when objects are added, etc.

The pages that are contained in a diagram are listed and handled according to a fixed order. While the order of pages is initially defined by the order in which the pages are added when created, pages can be renamed and rearranged with the *Edit* menu choice in the *Pages* menu.

This order is reflected in some of the menu choices that are related to pages. Also, the structure displayed by the Organizer will adopt the same order. See "The Organizer" on page 39 in chapter 2, *The Organizer*.

The Editor User Interface and Basic Operations

The Editor User Interface

The editor window can be used for viewing OM, SC, DP, MSC and HMSC diagrams.

The general user interface is described in chapter 1, *User Interface and Basic Operations*.

These functions are provided:

- The *Entity Dictionary Window*, which provides a name reuse facility as well as information for link endpoints.
- A symbol menu where you select the symbols that are to be inserted.
- A text window where you may edit text associated with symbols.

When you edit **OM** diagrams, three auxiliary windows are also provided:

- The *Browse & Edit Class Dialog* which you use to browse amongst and inspect and edit OM classes distributed across page and diagram boundaries.
- The *Line Details Window* which allows you to inspect and edit the various attributes of OM lines.
- The *Symbol Details Window* which allows you to inspect and edit OM symbol attributes.

When you edit **DP** diagrams, two of the auxiliary windows are also available:

- The *Line Details Window* which allows you to inspect and edit the various attributes of DP lines.
- The *Symbol Details Window* which allows you to inspect and edit DP symbol attributes.

When you edit **MSC** diagrams this function is also available:

- An instance ruler which allows you to view the kinds of instances, even if the instance head symbols are not currently in view.

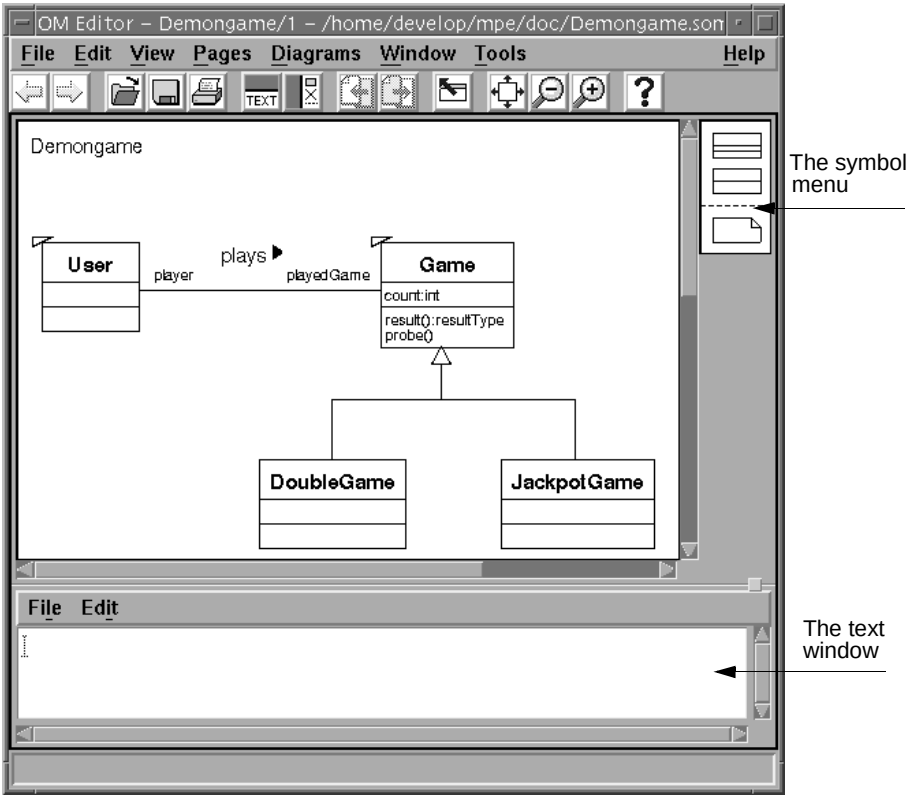


Figure 279: The editor window showing an OM diagram **(on UNIX)**

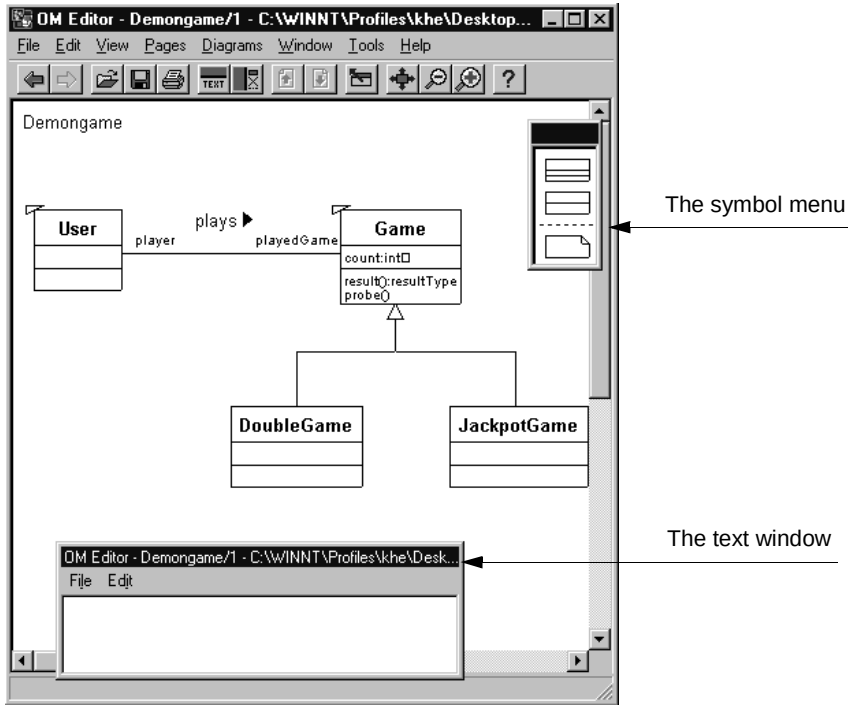


Figure 280: The editor window showing an OM diagram **(in Windows)**

The Editor Drawing Area

The drawing area is the part of the window which displays the symbols, lines and text that constitute a page (a diagram can contain multiple pages) or an MSC or DP diagram.

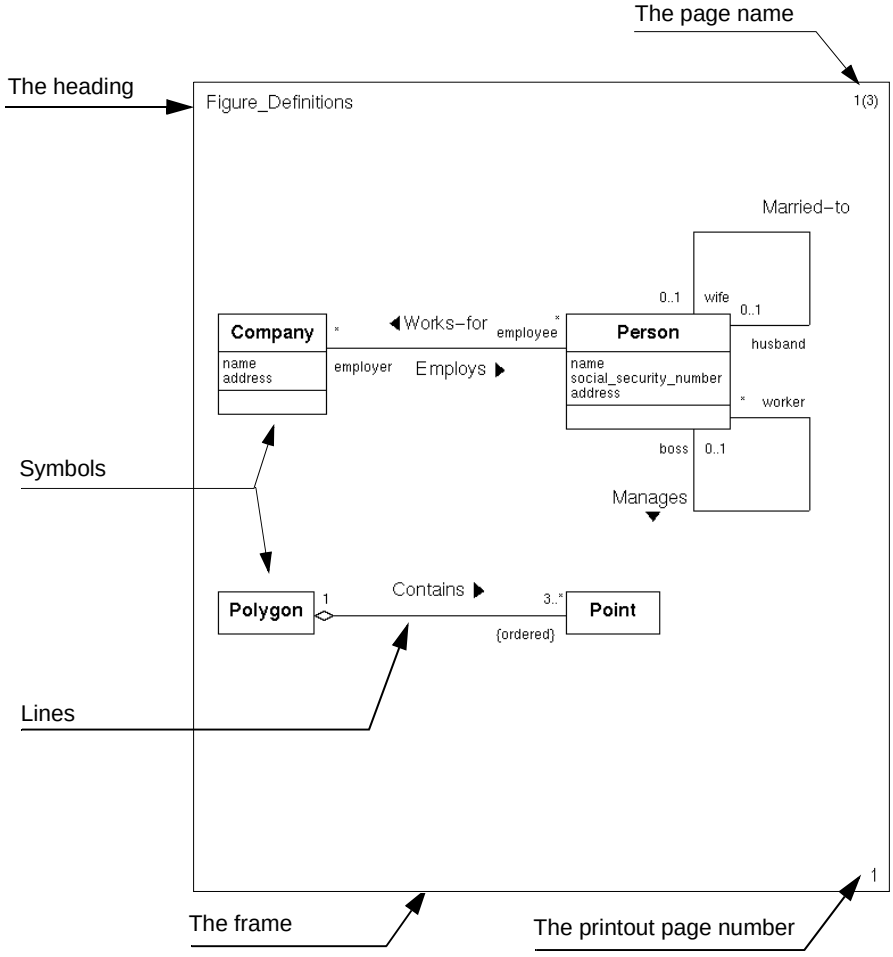


Figure 281: The drawing area with an OM page displayed

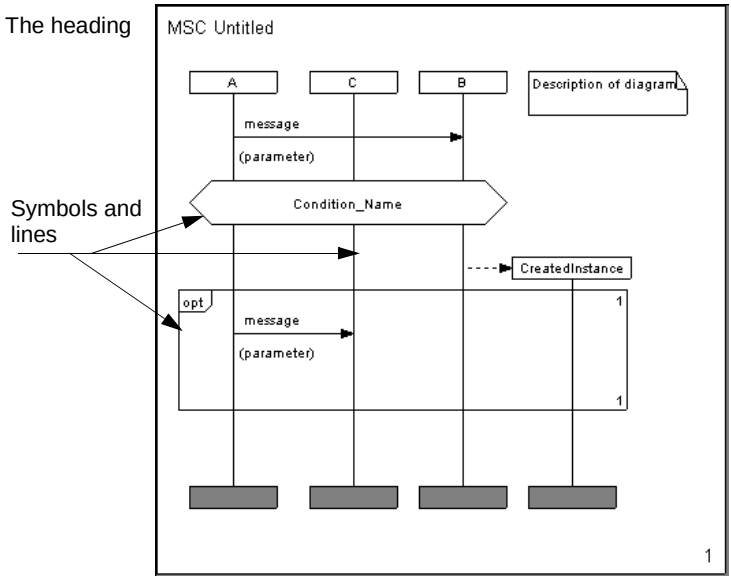


Figure 282: The drawing area with an MSC Diagram displayed

Drawing Area Boundaries

The drawing area is delimited by its boundaries, which correspond to the size of the page. No objects are allowed to be placed outside these boundaries. The drawing area uses a light background color, while the area outside the drawing area uses a grey pattern.

Within a diagram, each page has an individual size.

The Frame

The frame always coincides with the drawing area size. It is selectable but not editable. The frame is automatically selected with the *Select All* menu choice.

It is not possible to connect any diagram symbol to the frame. The frame only affects if the page is printed with or without a frame when printing selected objects, or when selected symbols are copied to a metafile **(Windows only)**.

The Heading

OM, SC and DP only: The heading contains the diagram name. The heading is editable but cannot be moved. The editor performs a textual syntax check on the name used in the heading, see "Diagram Name Syntax" on page 1732.

MSC and HMSC only: The heading identifies the chart type and name. The type cannot be edited.

The Page Name

All editors except MSC and DP.

In the upper right corner, the page name and the total number of pages in the diagram (within parentheses) are identified. The page name cannot be moved. To rename a page, use the *Edit* menu choice in the *Pages* menu.

The Printout Page Number

If a diagram page is larger than the paper format that is defined, the diagram page will be split into several printout pages. In this case, page numbers will be created. The page numbering follows a "down first, then right" fashion.

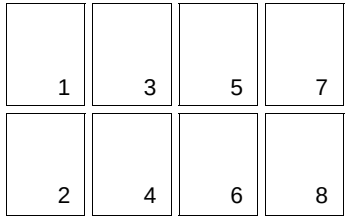


Figure 283: Page numbering in the editor

Grids

The editor uses two grids for an easy positioning of symbols, lines and textual elements:

- The symbol grid has a resolution of 5 * 5 mm, except for the **MSC** editor where it is 2.5 * 2.5 mm. All symbols adhere to the symbol grid.

- The line and text grid has a resolution of 2.5 * 2.5 mm. All lines and textual objects adhere to the line grid.

None of the grids can be changed.

Color on Symbols

All editors except MSC.

For each symbol type in the editor there is a preference for setting the color of the symbol. It is only the graphical part of the symbol and not the associated text(s) that will use the color setting. **On UNIX**, this setting is only valid on screen and all symbols will use the black color when printed on paper. **In Windows**, when using MSW Print the color settings will be sent to the printer as well. See "OM/SC/HMSC/MSC/DP Editor Preferences" on page 280 in chapter 3, *The Preference Manager*.

The text symbol is the same in OM, SC, DP and HMSC diagrams and has thus only one preference.

Keyboard Accelerators

In addition to the standard keyboard accelerators, the editor features the following:

Accelerator	Reference to corresponding command
`Ctrl+D`	"Next page" on page 1630
`Ctrl+U`	"Previous page" on page 1630
`Ctrl+T`	"Symbol Details" on page 1672
`<Delete>`	"Clear" on page 1670
`Ctrl+1`	"Show Organizer" on page 15 in chapter 1, *User Interface and Basic Operations*
`Ctrl+2`	"Connect to Text Editor" on page 1686

Quick-Buttons

In addition to the generic quick-buttons in all SDL Suite and TTCN Suite tools, the editor tool bar contains the following quick-buttons. See also "General Quick-Buttons" on page 24 in chapter 1, *User Interface and Basic Operations*.

Text window on / off
Toggle the text window between visible and hidden (see "Text Window" on page 1661).

Symbol menu on / off
Toggle the symbol menu between visible and hidden (see "Symbol Menu" on page 1632).

Previous page
(not valid for MSC and DP)
Open previous page in flow (similar to "<Page Name>" on page 1678). Dimmed if no previous page exists.

Next page
(not valid for MSC and DP)
Open next page in flow (similar to "<Page Name>" on page 1678). Dimmed if no next page exists.

Toggle Scale
Toggle the scale between a scale to show the complete diagram in the window (similar to *Overview* in "Set Scale" on page 1675) and a scale of 100%. For MSCs, the scale is adjusted to fit the diagram *width* in the window instead of the complete diagram.

Make space for new events
(MSC only)
Create space between two events (see "Make Space" on page 1673).

Remove space between two events
(MSC only)
Remove the unrequired space between two events.

Instance ruler on / off
(MSC only)
Toggle the instance ruler between visible and hidden (see "Instance Ruler" on page 1676).

Scrolling and Scaling

You can scroll the view vertically and horizontally by using the scroll-bars. The view may also be scrolled automatically when you move the cursor beyond the current view, for example when you move an object or add a symbol.

If you move the cursor close to the edge of the current view, the automatic scrolling is slow. If you move it further beyond, the scrolling is quicker.

You can scale the view by specifying a scale or by zooming in and out.

To specify a scale:

1. Select *Set Scale* from the *View* menu.
2. In the dialog that will be opened, you can either:
   - Use the slider for setting the scale and then click the *Scale* button.

   - Click the *Overview* button to adjust the drawing area to the size of the window. (This has the same effect as the *Scale Overview* quick-button.)

To zoom in or out:

- Click the quick-button for *zoom in* or *zoom out*.

Moving MSC Selection with Arrow Keys

You can move the selection using arrow keys in the MSC editor if there is one single symbol or line selected. The selection is moved along one instance in the vertical direction and between two instances in the horizontal direction.

In the MSC editor you are always in text editing mode, if there is a text to edit for the currently selected symbol. To move the symbol selection, press the arrow key one or several times in the direction you want to move. At first, only the text cursor position might be changed. But when the text cursor reaches the border of the text currently being edited (or

if there is no text being edited), then the symbol selection will be moved.

One shortcut is available for fast-forward moving to the next/previous condition or instance head symbol attached to the current instance: Shift+up or down arrow key.

Lock Files and Read-Only Mode

When you open a diagram or document file, e.g. `a.ssy`, a lock file `a.ssy.lck` is created. If another user tries to open the same diagram or document file before you close it, a dialog will appear informing the other user that the file is in use. There are two choices in the dialog:

- Open the file in read-only mode.
- Reset the lock and open the file in read-write mode. (This alternative should only be used if the existing lock file is obsolete.)

The read-write mode is the normal editing mode. In read-only mode, you are not allowed to make any changes to the diagram.

You will also enter read-only mode if you open a diagram file that you do not have write permission for. The read-only mode is indicated by the words *read-only* in the window title.

If you want to edit a diagram that is in read-only mode, there are two alternative actions:

- Change the file access rights in the file system to give you write permission for the diagram file. When that is done, you can change from read-only mode to read-write mode with the *Revert Diagram* operation.
- Select *File* > *Save As* to save the diagram in a new file with write permission.

Symbol Menu

The *symbol menu* contains the symbols that you can place into the drawing area.

On UNIX, the symbol menu is a fixed-sized, non-moveable auxiliary window, associated with the drawing area and placed to the right of it. Each editor window has its own symbol menu.

In Windows, the symbol menu is a fixed-sized, moveable window that can be placed anywhere on the screen, not necessary within the limits of the editor window. A single symbol menu is shared by all instances of the editor currently running.

The symbol menu can be made invisible and visible again with a menu choice, *Window Options*, or a quick-button. **In Windows:** When visible, the symbol menu will always be placed on top of the editor window, if the two windows overlap.

Basically, when you select a symbol in the symbol menu and click it into the drawing area, it is added to the diagram. This chapter does not describe how to work with symbols. Working with MSC symbols is described in chapter 41, *Editing MSC Diagrams*. See also "Working with Symbols" on page 1901 in chapter 43, *Using the SDL Editor*, as working with symbols in this editor is similar to working with symbols in the SDL Editor.

OM only: When a new class or object symbol is placed in the drawing area, it will automatically become an endpoint if the preferences AlwaysEndpointClass and AlwaysEndpointObject, respectively, are set.

The contents of the symbol menu depends upon the type of diagram that is displayed in the editor window, as can be seen in Figure 284. When you switch between diagrams of different types in the editor, the symbol menu changes accordingly.

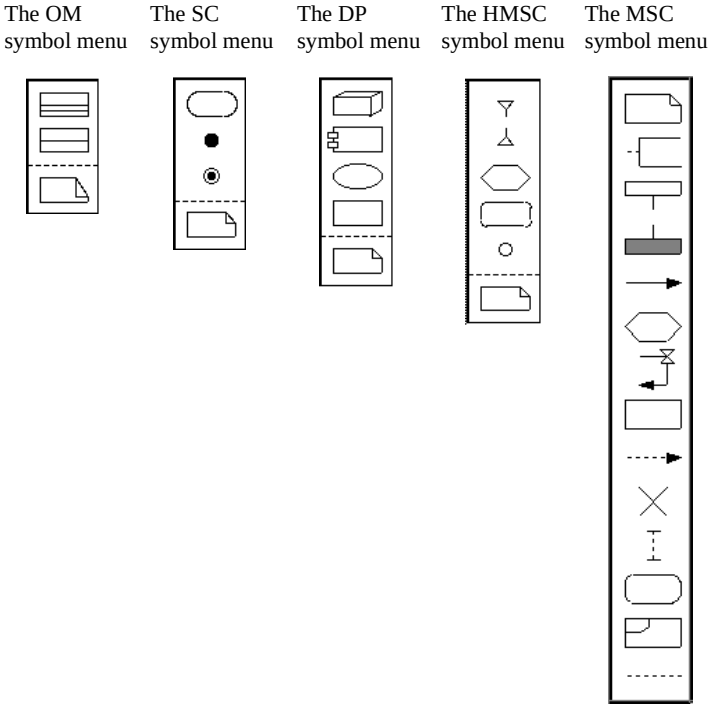


Figure 284: The editor symbol menus

About Symbols and Lines

The notation used for symbols and lines in the **OM Editor** complies with Static Structure (Class) diagrams defined in UML (Unified Modeling Language), version 1.3.

The notation used for symbols and lines in the **SC Editor** complies with State Chart diagrams defined in UML (Unified Modeling Language), version 1.3.

The notation used for symbols and lines in the **DP Editor** is based on the notation of Deployment and Component diagrams defined in UML (Unified Modeling Language), version 1.3.

The notation used for symbols and lines in the **HMSC Editor** complies with the ITU-T Z.120 recommendation from 1996. The comment symbol and the parallel frame symbol are not supported.

The notation used for symbols and lines in the **MSC Editor** complies with the ITU-T Z.120 recommendation from 1996. All MSC'96 symbols except the general ordering arrow are supported. Furthermore the MSC Editor does not support drawing of messages to the environment frame and message gates. Inline expressions are always global (i.e. they cover all instances).

For short explanations of when to use different symbols, see appendix 44, *Symbols and Lines – Quick Reference, on page 2053*.

Symbols

Diagrams contain different types of symbols that describe the structure of the diagram. All symbols must be placed inside the frame symbol. Figure 284 on page 1634 identifies these symbols in the symbol menu.

You can draw symbols in color, see “Color on Symbols” on page 1629.

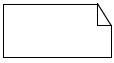
All symbols that are available in the symbol menu are selectable and moveable; they can be assigned arbitrary locations by you, with the exception that symbols are normally not allowed to overlap[1]. **SC only:** symbols can however be placed inside a state symbol, see “State Symbols” on page 1641.

1. In certain situations symbols will overlap as a consequence of automatic resizing when new text is entered in the symbols. Such symbols can only be moved to areas where an overlap does not occur.

Common Symbol

A symbol common for all the editors is:

Symbol Appearance	Symbol Name	Summary
	Text	Informal specification, global comments



Text Symbols

Text symbols contain informal text. It is not possible to connect any lines to a text symbol.

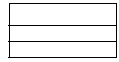
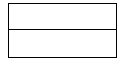
Double-clicking on a text symbol minimizes the symbol if not already minimized and maximizes the symbol if not already maximized.

Text symbols are typically used to add comments to diagrams or to convey system information (**HMSC Editor**: on a global level) that cannot be captured using the constructs offered by the editor.

When all of the text within a text symbol is in view, the upper right corner looks like a piece of paper that has been folded. When any portion of the text within a text symbol cannot be seen (because the text symbol is too small), the upper right corner looks like it has been clipped.

OM Symbols and Lines

See also appendix 44, *Symbols and Lines – Quick Reference, on page 2053*.

Symbol Appearance	Symbol Name	Summary
	Class	Class definition
	Object	Class instance

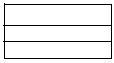
Text in Class and Object Symbols

OM symbols have one or more text compartments, which should be filled with a textual expression. While the OM Editor does not require that symbol text compartments should contain text, it will perform a syntactic check on text compartments as soon as they are changed and deselected.

The syntax rules of OM diagrams are detailed in "Object Model Syntax" on page 1731.

> **Note:**
>
> The OM Editor does not enforce syntax correctness in OM class and object symbols. This means that it is possible to use any conventions you desire for the textual contents of symbols; however, some features offered by the OM Editor will not be available, notably the *Browse & Edit Class* dialog.

Syntax checking on text in general is described in greater detail under "Textual Syntax Checks" on page 1660.



Class Symbols

A class symbol is divided into three horizontal compartments. In order from the top, the compartments are:

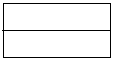
- The name compartment
- The attributes compartment
- The operations compartment

The text in these compartments is parsed by the OM Editor and should conform to the syntax specified in "Class Symbol Syntax" on page 1732.

In the name compartment two more extra text fields can appear. If adding text to the stereotype and properties fields in the Symbol Details window, these texts will also be visible inside the name compartment. These texts are not subject to syntactic checks.

Figure 285: Class symbol with stereotype and properties texts

If you double-click a class symbol, the *Browse & Edit Class* dialog will be opened (see "Browse & Edit Class Dialog" on page 1691).



Object Symbols

Used to describe an OM object, or an instance of a class.

An object symbol differs from a class symbol in that it does not contain an operations compartment. In addition, the remaining compartments require a somewhat different syntax, see "Object Symbol Syntax" on page 1734.

The object symbols handles stereotype and properties texts in the same way as the Class symbol.

If you double-click an object symbol whose name compartment includes a class name[1], the *Browse & Edit Class* dialog will be opened (see "Browse & Edit Class Dialog" on page 1691).

1. A class name is considered present only if the class name follows the object name after being separated by a colon ':' character.

Lines

Lines are the graphical objects that interconnect objects. Normally a line interconnects two symbols but it could also connect a symbol with another line. The following types of lines interconnecting symbols are defined in OM diagrams (see Figure 286):

- Association
- Aggregation
- Generalization

The only line interconnecting a symbol and a line in OM diagrams is:

- Link Class

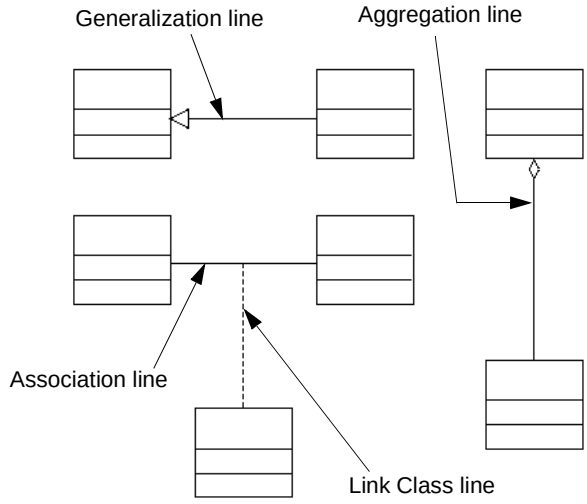


Figure 286: Lines in OM diagrams

To insert lines, select a class or object symbol, drag one of the *handles* that appear on the source symbol and connect it to the target symbol. There is one handle for each type of line.

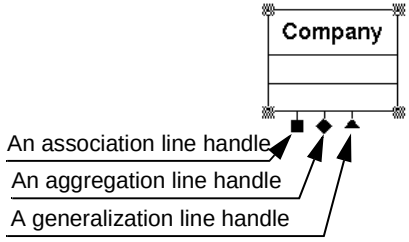


Figure 287: Handles for different line types

The Link Class line handle is placed on selected association and aggregation lines.

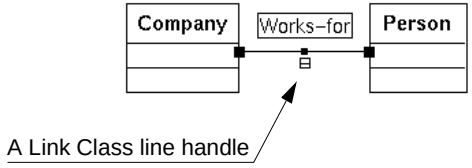


Figure 288: Handle for Link Class line

When a handle is clicked, the status bar shortly describes what the handle is used for and how to draw a line. This chapter does not describe how to work with lines. However, working with lines in the OM Editor is similar to working with lines in the SDL Editor; see "Working with Lines" on page 1932 in chapter 43, *Using the SDL Editor*.

Lines are always connected to objects; they are not allowed to exist on their own. A line is allowed to overlap any other object.

Lines are selectable; they can be moved and reshaped by you. Some layout work is performed automatically by the OM Editor.

Double-clicking on a line brings up the Line Details window (see "Line Details Window" on page 1696).

See also "Line Attribute Objects" on page 1653.

SC Symbols and Lines

See also appendix 44, *Symbols and Lines – Quick Reference, on page 2053*.

Symbol Appearance	Symbol Name	Summary
	State	State definition.
	Start	Starting point of the SC.
	Termination	Termination point of the SC.

Text in State Symbols

State symbols have three text compartments which should be filled with textual expressions, but are allowed to remain empty. A syntactic check on a text compartment is performed as soon as it is changed and deselected.

State Symbols

A state symbol contains a state section which is divided into horizontal compartments. In order from the top, the compartments are:

- The Name compartment
- The Internal Activity compartment

The text in these compartments should conform to the syntax specified in "State Symbol Syntax" on page 1736.

The state symbol may have an additional compartment with a graphic region holding a nested SC. This is called the *substate* compartment and states with this compartment are often called *hierarchical states* or *superstates*. The state will be extended with this compartment when a symbol is placed within it. The compartment disappears when the last symbol is removed from it.

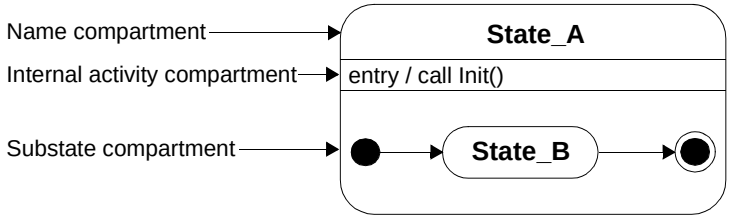


Figure 289: A state with its compartments

Text symbols are allowed to be placed inside the substate compartment but will not belong to the state. Thus, it will not automatically be part of the operations done on the state symbol, e.g. move, copy, etc.

Start Symbols

The start symbol is used to indicate where a SC starts.

Termination Symbols

The termination symbol is used to indicate where a SC terminates.

Lines

Lines are the graphical objects that interconnect objects. There is only one kind of line interconnecting symbols in SC diagrams, a transition line (see Figure 286):

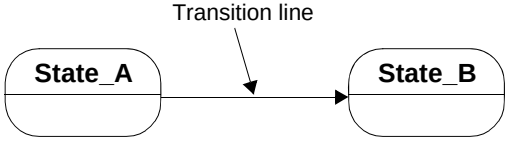


Figure 290: Transition line in SC diagrams

To insert lines, select a start or state symbol, drag the *handle* that appears on the source symbol and connect it to the target symbol.

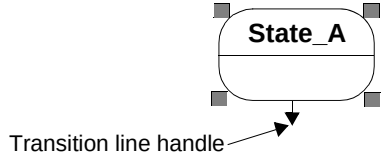


Figure 291: Handle for transition lines

When a handle is clicked, the status bar shortly describes what the handle is used for and how to draw a line. This chapter does not describe how to work with lines. However, working with lines in the SC Editor is similar to working with lines in the SDL Editor; see "Working with Lines" on page 1932 in chapter 43, *Using the SDL Editor*.

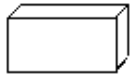
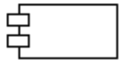
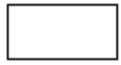
Lines are always connected to objects; they are not allowed to exist on their own. A line is allowed to overlap any other object.

Lines are selectable; they can be moved and reshaped by you. Some layout work is performed automatically by the SC Editor.

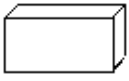
See also "Line Attribute Objects" on page 1653.

DP Symbols and Lines

See also appendix 44, *Symbols and Lines – Quick Reference, on page 2053*.

Symbol Appearance	Symbol Name	Summary
	Node	A run-time physical object that represents a computational resource (see [6] on page 1767).
	Component	A physical, replacable part of a system that packages implementation (see [6] on page 1767).
	Thread	An OS thread, i.e. a light-weight process. Not to be mistaken for an OS task, i.e. a heavy-weight process.
	Object	An SDL entity, i.e. a system, block or process.

Double-clicking on a DP symbol brings up the Symbol Details window (see "Symbol Details Window" on page 1714).



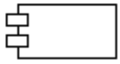
Node Symbols

For node symbols the following text attributes can be edited:

- The name text
- The stereotype text
- The properties text

The name text can be edited directly in the symbol and should conform with the syntax described in "Node Symbol Syntax" on page 1739. The other texts can be edited in the Symbol Details window and are not subject to syntactic checks.

Figure 292: Node symbol with stereotype and properties texts



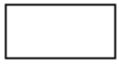
Component Symbols

The text attributes of component symbols are the same and follow the same rules as those of node symbols. You can select the integration model for a component symbol from the Symbol Details dialog box. The syntax rules that apply are described in "Component Symbol Syntax" on page 1739.

Figure 293: Component symbol with properties text

Thread Symbols

For thread symbols only the name text is visual. From the Symbol Details dialog box, you can edit thread priority, stack size, queue size and max signal size for a thread. The syntax rules described in "Thread Symbol Syntax" on page 1739 are applied.



Object Symbols

Object symbols have the following text attributes:

- The name text
- The stereotype text
- The properties text
- The qualifier text

In the Symbol Details window the stereotype, properties and qualifier texts can be added. If the stereotype text for an object symbol is **`process`** the appearance of the symbol is changed.

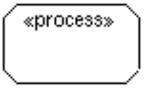


Figure 294: Object with stereotype "process"

The qualifier of an object symbol is displayed in the status bar when pointing the mouse at the symbol.

The syntax rules for the object symbol text attributes are described in "Object Symbol Syntax" on page 1739.

Lines

Lines are graphical objects that interconnect objects. Two types of lines are defined in DP diagrams (see Figure 295):

- Association
- Composite Aggregation

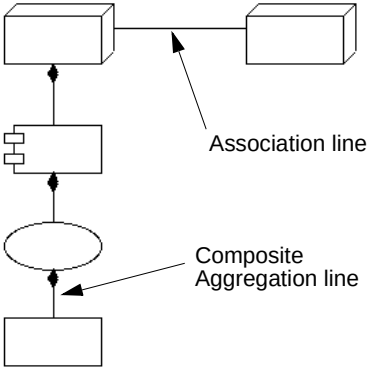


Figure 295: Lines in DP diagrams

To insert lines, select a symbol, drag one of the *handles* that appear on the source symbol and connect it to the target symbol. There is one handle for each type of line.

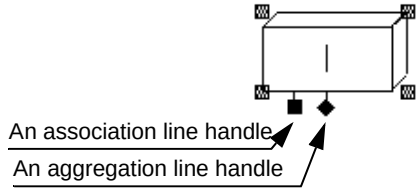


Figure 296: Handles for different line types

When a handle is clicked, the status bar shortly describes what the handle is used for and how to draw a line. This chapter does not describe how to work with lines. However, working with lines in the DP Editor is similar to working with lines in the SDL Editor; see "Working with Lines" on page 1932 in chapter 43, *Using the SDL Editor*.

Lines are always connected to objects; they are not allowed to exist on their own. A line is allowed to overlap any other object.

Lines are selectable; they can be moved and reshaped by you. Some layout work is performed automatically by the DP Editor.

Double-clicking on a line brings up the Line Details window (see "Line Details Window" on page 1711).

See also "Line Attribute Objects" on page 1653.

HMSC Symbols and Lines

See also appendix 44, *Symbols and Lines – Quick Reference, on page 2053*.

Symbol Appearance	Symbol Name	Summary
	MSC Reference	Reference to another MSC diagram or MSC reference expression
	Condition	Expresses alternative path in an HMSC
	Connection Point	Expresses that lines are connected at this point
	Start	Start of the HMSC road map.
	End	End of the HMSC road map.

Reference Symbols

The reference symbol is used to refer to other (H)MSC's of the MSC document. The MSC references are objects of the type given by the references MSC.

MSC references may not only refer to a single MSC, but also to MSC reference expressions. MSC reference expressions are textual MSC expressions constructed from the operators **alt**, **par**, **seq**, **loop**, **opt**, **exc** and **subst**, and MSC references.

The text is parsed by the HMSC Editor and should conform to the syntax specified in "Reference Symbol Syntax" on page 1742.

Double-clicking on a reference symbol opens the referenced MSC diagram. Note that if a reference symbol contains more than one name, you should place the cursor directly on one of the MSC names; otherwise an dialog is presented, see "Navigate" on page 1688.

Condition Symbols

The condition symbol in HMSC's can be used to indicate global system states and impose restrictions on the MSC's that are referenced in the HMSC.

The text is parsed by the HMSC Editor and should conform to the syntax specified in "Condition Symbol Syntax" on page 1741.

Connection Point Symbols

The connection point symbol in HMSC's is used to indicate that two (or more) crossing lines are actually connected.

Start Symbols

The start symbol in HMSC's is used to indicate where an HMSC road map starts.

End Symbols

The end symbol in HMSC's is used to indicate where and HMSC road map ends.

Lines

Lines are the graphical objects that interconnect objects. There is only one kind of line in the HMSC Editor.

To insert lines, select any[1] of the symbols, drag the *handle* that appear on the source symbol and connect it to the target symbol.

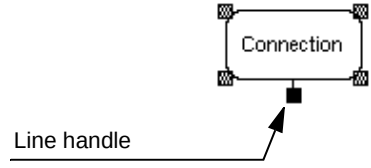


Figure 297: Symbol line handle

1. This does not apply to the stop symbols, since lines can not be drawn from it, only to it.

When a handle is clicked, the status bar shortly describes how to draw a line. This chapter does not describe how to work with lines. However, working with lines in the HMSC Editor is similar to working with lines in the SDL Editor; see "Working with Lines" on page 1932 in chapter 43, *Using the SDL Editor*.

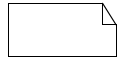
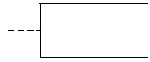
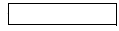
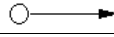
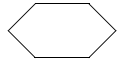
Lines are always connected to objects; they are not allowed to exist on their own. A line is allowed to cross (but not overlap) another line. Since lines in HMSC's are directed, lines always starts at the bottom of a symbol and ends on the top of a symbol.

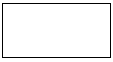
Lines are selectable; they can be moved and reshaped by you.

MSC Symbols and Lines

See also appendix 44, *Symbols and Lines – Quick Reference, on page 2053*.

Symbols are the graphical objects that build up the contents of an MSC. Lines are the graphical objects that interconnect symbols. Lines are available in the symbol menu and are thus handled as symbols.

Symbol Appearance	Symbol Name	References to Z.120
	Text	Z.120 2.3
	Comment	Z.120 2.3
	Instance head	Z.120 4.2
	Instance end	Z.120 4.2
	Message	Z.120 4.3
	Message-to-self	Z.120 4.3
	Found message	Z.120 4.3
	Lost message	Z.120 4.3
	Condition	Z.120 4.6
	Timer[a]	Z.120 4.7
	Separate timer set[b]	Z.120 4.7
	Separate timer reset	Z.120 4.7

Symbol Appearance	Symbol Name	References to Z.120
	Separate timer consumed / timeout	Z.120 4.7
	Action	Z.120 4.8
	Create	Z.120 4.9
	Stop	Z.120 4.10
	Coregion	Z.120 5.1
	MSC Reference	Z.120 5.4
	Inline expression	Z.120 5.3
	Inline separator	Z.120 5.3

a. The Z.120 symbols *timer set* and *time-out* are in this case merged into one symbol, the MSC Editor timer symbol.

b. A separate timer reset or timer consumed symbol is created by creating a separate timer set symbol, followed by changing its status to either reset or consumed.

Syntax Checking on Symbols and Lines

The MSC Editor checks that the **symbols** are positioned in the MSC in accordance to rules governed by Z.120. Lines are always connected to at least one symbol, they are not allowed to exist on their own. The MSC Editor checks that **lines** are connected to symbols in accordance to Z.120.

Graphical Properties of Symbols and Lines

All symbols that are available in the symbol menu are selectable and moveable. They can be placed automatically or you may, as long as the Z.120 syntax rules are respected, assign them arbitrary locations.

Lines are selectable; you can move them and reconnect them. Some layout work is performed automatically when a line is drawn.

Most of the symbols are not resizable; these are indicated by grayed selection squares. Other symbols may be resized; this is shown by a filled selection square.

Textual Attributes

Textual objects are the textual attributes that are related to a symbol or a line.

Some MSC symbols have one or multiple text attributes. A text attribute should be filled with an MSC/PR expression (textual expression) that is syntactically correct according to Z.120, alternatively filled with some informal text if the MSC concept is used informally.

Text attributes related to messages and timers may be moved freely by you. Textual objects are allowed to overlap any other objects.

Line Attribute Objects

This section applies to OM, SC and DP only.

Line attribute objects represent additional items of information associated with lines. It is not possible to create free line attribute objects that are not associated with any line; line attributes are always associated to, and destroyed with, their respective lines.

SC only: Each transition line has a line attribute object, the *transition label*, which is pre-created by the SC Editor when you create a line. You may ignore this attribute or fill in the textual contents, as appropriate. The text should conform to the syntax specified in "Transition Line Syntax" on page 1737.

OM only: Each type of line has a primary line attribute object, the *name* (for associations and aggregations) or *discriminator* (for generalizations) attribute, which is pre-created by the OM Editor when you create a line. You may ignore the primary attribute or fill in the textual contents, as appropriate.

DP only: Each type of line has a primary line attribute object, the *protocol* (for associations) or *multiplicity* (for aggregations) attribute, which is pre-created by the DP Editor when you create a line. You may ignore the primary attribute or fill in the textual contents, as appropriate.

OM and DP only: A number of secondary line attribute objects can be created for each line using the Line Details window (see "Line Details Window" on page 1696 and "Line Details Window" on page 1711 respectively). The exact number and types of line attribute objects depends on the type of line and is summarized in tables below.

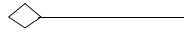
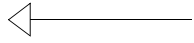
Line attribute objects have a number of common characteristics:

- They are conceptually attached to the closest line segment of the associated line and will be moved and redrawn as appropriate whenever the line's layout is changed[1].
- A selected textual line attribute object is indicated by a small rectangle which shrink-wraps the actual text. If the textual attribute is editable, the text cursor will appear at the current insertion position inside the text, and the text window will contain the text of the selected attribute object.
- Textual line attribute objects always display their entire contents; it is not possible to collapse them to avoid displaying large amounts of text, as is the case with class and object symbols.
- **OM and DP only:** Syntax checks are not performed on textual line attribute objects.
- **OM and DP only:** All textual line attributes except the line's primary attribute will be destroyed if their textual contents is cleared and the attribute is selected. To recreate a destroyed attribute, use the Line Details window.

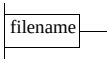
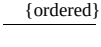
OM only: While most line attributes can be moved freely and are allowed to overlap other objects, some attributes, such as the association qualifier, are restrained by graphical layout restrictions and cannot be freely moved.

1. While the results are normally what you would prefer, it is always possible to manually adjust the position of moveable line attributes after repositioning the line.

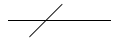
OM Line Attribute Objects

Name and Graphical Appearance	Line Attribute Objects (• Primary and – Secondary)
Association	• Name of the association – Reverse Name – Role Name – Role Multiplicity – Qualifier – Sorted – Ordered – Derived Attribute – Constraint
Aggregation 	• Name of the aggregation – Reverse Name – Role Name – Role Multiplicity – Qualifier – Sorted – Ordered – Composite
Generalization 	• Discriminator of the generalization
Link Class	• *No line attributes*

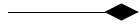
OM Aggregation/Association Attributes[1]

Attribute Appearance	Attribute Name (Properties)	Summary
	Name (Pre-created, Editable, Optional arrow)	Names a conceptual connection between object instances. While always bidirectional, the name usually denotes traversal of the association in the "forward" direction, as defined by the optional arrow.
	Reverse Name (Editable, Optional arrow)	Allows a separate name to be used when the link is traversed in the opposite direction.
	Qualifier (Editable, Not moveable)	Describes an association between two objects that is defined by the identity of a third object.
	Ordered (Not editable)	The *ordered* attribute puts additional constraints on a one-to-many association.
	Sorted (Not editable)	The *sorted* attribute puts additional constraints on a one-to-many association.
	Role Name (Editable)	Each end of an association is called a role. While redundant, well-named roles can sometimes facilitate analysis.
	Role Multiplicity (Editable)	Multiplicity is used to constrain the number of related objects. Multiplicity can be fixed, a closed range, an open range, or any combination of these.

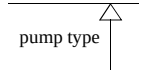
1. While these attributes are available both for associations and aggregations, they are much more often used for associations. The use of these attributes for aggregations depends on how sharply the analysis or design separates between association and aggregation.

Attribute Appearance	Attribute Name (Properties)	Summary
	Derived Attribute (Not editable, Not moveable)	Indicates that the association represents redundant information, that can be calculated by following other associations.
	Constraint (Editable)	Allows the specification of arbitrary constraints. The specified text is automatically placed between '{' and '}'.

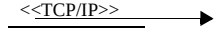
OM Aggregation Attributes

Attribute Appearance	Attribute Name (Properties)	Summary
	Composite (Editable)	Indicates aggregation with strong ownership.

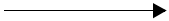
OM Generalization Attributes

Attribute Appearance	Attribute Name (Properties)	Summary
	Discriminator (Pre-created, Editable)	The discriminator describes which property of an object is being abstracted by a particular generalization relationship.

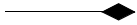
SC Line Attribute Objects

Name and Graphical Appearance	Line Attribute Objects
Transition 	• Transition label of the transition

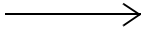
SC Transition Attributes

Attribute Appearance	Attribute Name (Properties)	Summary
event (arguments) [guard-condition] /actions 	Transition label (Pre-created, Editable)	Describes the transition. For a description of the syntax, see "Transition Line Syntax" on page 1737.

DP Line Attribute Objects

Name and Graphical Appearance	Line Attribute Objects (• Primary and – Secondary)
Aggregation 	• Name of the association – Protocol – Encoding – Direction
Composite Aggregation 	• Multiplicity of the composite aggregation

DP Association Attributes

Attribute Appearance	Attribute Name (Properties)	Summary
	Protocol (Pre-created, Editable)	The protocol of the communication path.
name 	Name (Editable)	The name of the communication path.
{BER} 	Encoding (Editable)	The encoding of the communication path.
	Direction (Editable)	The direction of the communication path.

DP Aggregation Attributes

Attribute Appearance	Attribute Name (Properties)	Summary
3,*	Multiplicity (Pre-created, Editable)	Multiplicity is used to constrain the number of related objects. Multiplicity can be fixed or an infinite number.

Editing Text

General

The drawing area is a container for different types of objects that each may contain zero or more *text compartments*. In addition, the lines that connect objects also have zero or more *textual attribute objects*.

The editor allows these textual objects to be edited directly in the drawing area, allowing you to see directly how the textual object resizes to shrink-wrap the text.

To directly edit a textual object in the drawing area, simply position the text insertion cursor at the desired position in the text. To indicate that a textual object is being edited, a thin vertical bar designating the text insertion cursor appears at the selected text insertion point. As soon as you type some text, the keyboard input mode will change to *text editing mode*. This is indicated by a flashing and thicker insertion bar.

In text editing mode, all accelerator keys are interpreted as input to the text. As an example, the `<Delete>` key will in text editing mode delete a character, but in non-text editing mode it will remove (*Clear*) an entire symbol. This means that you can only use text editing keyboard accelerators like `<Home>`, `<End>`, `<Ctrl+A>` and `<Ctrl+E>` in text editing mode only.

While useful for quickly editing small texts, direct editing in the drawing area suffers from some restrictions:

- While it is possible to position the insertion cursor in the text using the mouse or the arrow keys, it is not possible to make any selections in the text, for deletion or replacement.

- If the text becomes large enough to cross the drawing area boundary, parts of the text will no longer be visible.

To alleviate these and other limitations, the editor offers a *text window* that provides a more complete set of text editing features. See "Text Window" on page 1661 for more information. In addition, it is possible to use an external text editor for larger amounts of texts; see "Connect to Text Editor" on page 1686.

Regardless of whether editing takes place directly in the drawing area or in the text window, the editor makes sure that the contents of both displayed texts are consistent, which makes it convenient to use both text editing mechanisms in the same diagram.

Textual Syntax Checks

Some texts in the editor are subject to syntactic checks as soon as they are changed. Errors detected during syntax checks will be displayed in the textual object by underlining the first characters at the position where the first syntax error was detected.

Minimum Size of Textual Symbols

While symbols are resized automatically to fit the containing text, the editor defines a minimum size for each symbol, ensuring that even empty symbols will be recognizable:

- The predefined minimum width of any symbol with text is 20 mm.
- The minimum height for symbols with text is calculated based on the height of the symbol font.
- The minimum height for text symbols is 10 mm.

Delayed Updating of the Drawing Area

When you edit the text in the text window, the updating of the drawing area is at least delayed for the value of the preference FontText*MinimumTextUpdateDelay.

If the text is large, the update delay is extended proportionally. The text window is however always updated immediately.

In addition, any attempt to enter an illegal character will always result in an audible warning.

Text Window

The text window works in the same way for OM, SC, DP, MSC and HMSC diagrams. There is one text window common for OM, SC, DP and HMSC diagrams, and a separate text window for MSC diagrams.

On UNIX, the text window is a pane of the editor window and can be resized vertically.

In Windows, the text window is a resizable and moveable window that can be placed anywhere on the screen, not necessary within the limits of the editor window. A single text window is shared by all instances of the editor currently running.

If you select one object in the editor window, the text window is updated to contain the text associated with that object, but not if you select more than one text object.

Each line (except for the last line) in the text window is terminated by a carriage return, and may consist of any number of legal characters.

The text window provides a menu bar with two menus which are described in:

- "File Menu of the Text Window" on page 1690
- "Edit Menu" on page 1669

Hiding and Showing the Text Window

You can hide and show the text window with the *Window Options* command from the *View* menu, or by using a quick-button.

In Windows: When visible, the text window will always be placed on top of the editor window.

Searching and Replacing Text in an MSC

The text window provides standardized functions for searching and replacing text in an MSC.

If all the textual objects you want to search in are not visible (i.e. *instance name*, *instance kind*, *instance composition*, *message name* and *message parameters*) adjust the *Diagram Options*.

Copying, Cutting and Pasting Text

The text window provides standardized clipboard functions for copying, cutting and pasting text between different **symbols, lines and text attributes**. These functions do not interfere with the clipboard functions for cutting, copying and pasting **objects**.

Programmable Function Keys (UNIX only)

On UNIX, it is allowed to tie a function key to a defined text string. When typing that defined function key, the programmed text string will be inserted at the current cursor location. You can customize your own programming of function keys.

Global X Resources

The function keys are set up as X resources. It is possible to set up both system default and user-defined X resources, allowing you to customize your environment. The X resources are defined in a file that is common for all users, namely

```
/usr/lib/X11/app-defaults/SDT
```

To program the function keys, insert the following lines anywhere into the `SDT` file:

```
/* Any suitable comment */
SDT*XmText.translations: #override \n\
<Key>F1: insert-string("F1Text") \n\
<Key>F2: insert-string("F2Text") \n\
<Key>F3: insert-string("F3Text") \n\
<Key>F4: insert-string("F4Text") \n\
<Key>F5: insert-string("F5Text") \n\
<Key>F6: insert-string("F6Text") \n\
<Key>F7: insert-string("F7Text") \n\
<Key>F8: insert-string("F8Text") \n\
<Key>F9: insert-string("F9Text")
/* Note the absence of \n\ on line 9 */
```

Note:

Omitting to define some of the function keys is permissible.

User-Defined X Resources

You can define your own function keys. This is done by defining the X-resources described above in a personal copy of the definition file and to store that file into your home directory:

```
~username/SDT
```

Alternatively, any directory designated by environment variable `XAPPLRESDIR` can be used.

Restrictions

1. Only one line for each function key can be defined. Attempting to define more than one line into one function key may cause an unpredictable result when pressing that key.

   For instance, it is not certain that the following line will produce the expected result:

   ```
   <Key>F1: insert-string("F1Line1\nLine2") \n\
   ```

2. Only the keys F1–F9 can be defined.

Changing Fonts on Text Objects

You may change the font faces and font sizes used in the textual objects displayed by the editor. All textual objects use the same font faces and font sizes, meaning that they cannot be changed individually and cannot be changed during an editor session.

The font faces which are available depend on the target system on which you are running the SDL Suite.

Defining What Font to Use

To modify the desired font size and font face, you must use the Preference Manager. See chapter 4, *Managing Preferences*.

Textual Objects Preferences

When the setting is in effect, the SDL Suite diagram editors will use the font face names given by the preference settings

OME*ScreenFontFamily
OME*PrintFontFamily

to select font face names. Note that in this way you can select different font names for screen and for print.

On UNIX, if you leave the Editor*FontText*ScreenFontFamily preference setting empty, you will edit your documents using the *SDT Draft* font, but print them using the font you specified with the Editor*FontText*PrintFontFamily setting.

Supported Font Faces

On UNIX, the availability of font faces is determined by the version of the X Windows server which is running. With revision 5 or higher (X11 R5), scalable fonts are supported. In that case, the available list of predefined font faces would be:

- Times
- Helvetica
- Courier
- SDT Draft (mapped to the Schumacher font)
- Other (mapped to a user-defined font)

In Windows, the availability of font faces is determined by the True-Type fonts that are currently installed on the computer (use for instance the Windows Control Panel to determine what is available).

Default Font Face

The default font face is Helvetica (see the preferences ScreenFontFamily and PrintFontFamily described in "OM/SC/HMSC/MSC/DP Editor Preferences" on page 280 in chapter 3, *The Preference Manager*).

On UNIX, if scalable fonts are not supported, the font face will be replaced by a *Schumacher* font which may be used in all circumstances **(MSC only)**.

Default Font Size

Font sizes are described in "NameTextHeight" on page 281 and "TextHeight" on page 282 in chapter 3, *The Preference Manager*.

They are used as follows:

Common Font Sizes

Text Object	Font Size	Other
Heading symbol	NameTextHeight	
Text symbol	TextHeight	

OM Font Sizes

Text Object	Font Size	Other
Class symbol name	NameTextHeight	Bold
Object symbol name	NameTextHeight	Bold
Stereotype text	TextHeight	
Properties text	TextHeight	Italic

SC Font Sizes

Text Object	Font Size	Other
State symbol name	NameTextHeight	Bold

DP Font Sizes

Text Object	Font Size	Other
Symbol name	NameTextHeight	Bold
Stereotype text	TextHeight	
Properties text	TextHeight	Italic

HMSC Font Sizes

Text Object	Font Size
Reference Symbol	TextHeight
Condition symbol	TextHeight

MSC Font Sizes

Text Object	Font Size
All MSC special symbols	TextHeight

Determining Which Scalable Fonts Your Server Can Access (UNIX only)

On UNIX, use the `xlsfonts` command to list installed fonts. Font names containing 0 for width and height are scalable.

Example 313: How to determine which fonts are available

From the OS prompt, typing:

```
hostname% xlsfonts | grep "\-0\-0\-" | more
```

will return a list of accessible scalable fonts.

Scalable Fonts Under R5 Servers

To use scalable fonts under X11R5 you must normally first connect to a font server.

Example 314: How to Start the Font Server

1. Start the font server on any local host:

   ```
   hostname% fs
   ```

2. Connect the server to `fs` indicating which host the font server is running on (which can be the same host that the X server is running on):

   ```
   hostname2% xset +fp tcp/<hostname>:7000
   ```

For further information see the X11R5 documentation or use `man fs` to read the manual page describing the font server you are running.

Disabling Font Scaling (UNIX only)

On UNIX, if the fonts look poor on the screen, a possible work-around is to disable the scaling option.

> **Note:**
> Disabling font scaling effectively disables WYSIWYG!

To do this, you should edit the `SDT` resource file.

1. Open the file `SDT` in a text editor.
2. Locate the line with the text: `SDT*sdtUseScalableFonts`
3. Change the line to `SDT*sdtUseScalableFonts: false`
4. Save the file and restart the SDL Suite environment.

Menu Bars

The editor menu bar provides the following menus:

- *File Menu*
- *Edit Menu*
- *View Menu*
- *Pages Menu*
- *Diagrams Menu*
- *Window Menu*
- *Tools Menu*
- *Help Menu*

The Help menu is described in "Help Menu" on page 15 in chapter 1, *User Interface and Basic Operations*.

File Menu

The *File* menu supports the following commands on diagrams:

- *New*
- *Open*
- *Save*
- *Save As*
- *Save a Copy As*
- *Save All*
- *Close Diagram*
- *Revert Diagram*
- *Print*
- *Exit*

The *File* menu choices are described in "File Menu" on page 8 in chapter 1, *User Interface and Basic Operations*, except *Print* which is described in *"The Print Dialogs in the SDL Suite and in the Organizer" on page 316 in chapter 5, Printing Documents and Diagrams*.

Edit Menu

The *Edit* menu provides editing functions that you can perform on objects in a diagram, or on text in the text window. The *Edit* menu contains the following menu choices:

- *Undo*
- *Cut, Copy and Paste*
- *Paste As*
- *Clear*
- *Clear Objects*
- *Keep Objects*
- *Expand/Collapse*
- *Line Details*
- *Symbol Details*
- *Class*
- *Drawing Size*
- *Select All*
- *Redirect*
- *Connect*
- *Status*
- *Make Space*
- *Decompose*

Undo

This command restores the content of the drawing area to its state prior to the most recently performed operation. Text editing operations cannot be undone.

Editor	Operations you can undo
OM, SC, DP, HMSC	• *Cut, Copy and Paste* and *Clear* • Resizing a page • Adding/moving symbols • Drawing/reshaping/redirecting lines
OM	• Editing a symbol (Symbol Details Window) • Editing a line (Line Details Window)
DP	• Editing a symbol (Symbol Details Window) • Editing a line (Line Details Window)

Editor	Operations you can undo
OM	• Editing a class (Browse & Edit Class Dialog) **Note:** Undoing an Edit Class operation will undo the changes in all affected diagrams.
SC	• Expanding/collapsing symbols
HMSC	• Editing a symbol • Editing a line
MSC	• *Cut, Copy and Paste* and *Clear* • *Undo* • *Redirect* • *Connect* • *Moving Objects* • *Adding and Removing Objects*

Cut, *Copy* and *Paste*

These commands provide standardized clipboard functions for copying, cutting and pasting objects in the drawing area or text in the text window.

You can interrupt a *Paste* operation by pressing `<Esc>`.

The **MSC** specific rules regarding the *Paste* command are described in "Pasting in MSC Diagrams" on page 1717.

Paste As

Pastes the currently copied object (from the OM or Text Editor) as an OM symbol or MSC object in the drawing area. The object is transformed and a link is optionally created between the copied and pasted objects.

The Paste As dialog is opened. See "The Paste As Command" on page 448 in chapter 9, *Implinks and Endpoints*.

Clear

This command removes the selected objects from the drawing area, or the selected text from the text window.

Also see "Deleting an Object" on page 461 in chapter 9, *Implinks and Endpoints*.

Clearing an *instance head* symbol or its *instance axis* line in an MSC diagram will also remove all objects that were connected with the instance axis regardless of if they were selected or not.

- A created instance left without "parent" is kept in the chart as static instance.
- Clearing an instance end reconnects the instance axis to the bottom of the chart.

Clear Objects

This **MSC** command removes all objects similar to the selected objects. For instance, select one message A object and one message B object and invoke this command to remove all occurrences of message A and B. To remove the empty spaces after removed objects, use *Rearrange*.

Keep Objects

This **MSC** command removes all objects of the same type as the selected objects that are not similar to the selected objects. For instance, select one message C object and one message D object and invoke this command to remove all messages except C and D. To remove the empty spaces after removed objects, use *Rearrange*.

Expand/Collapse

If a text symbol or a comment symbol is selected, this command allows you to expand or collapse the symbol from or to its minimum size.

If an MSC diagram is selected, the *Collapse* command collapses the selected instance axes to give you an overview of the diagram. A complex MSC can in this way be viewed in several decomposed MSCs. Two diagrams are created; one where the instances you have selected are collapsed into a decomposed instance and one where the behavior of the instances you have selected are shown. You can *Navigate* from a decomposed instance to the referenced diagram.

If an OM class, OM instance or SC state symbol is selected, this command selects whether all compartments should be displayed or not. A collapsed symbol will only display the name compartment (including

the stereotype and properties texts for an OM symbol), which is particularly useful when referring to a symbol that is defined elsewhere.

Line Details

This **OM and DP** command brings up the modeless Line Details window, that is used to inspect and edit the properties of the currently selected line. If there is none or more than one selected object, the items in the Line Details window will be dimmed.

The Line Details window and its appearance for different type of OM lines is described in "Line Details Window" on page 1696. For DP lines it is described in "Line Details Window" on page 1711.

Symbol Details

This **OM and DP** command brings up the modeless Symbol Details window, that is used to inspect and edit the stereotype and properties texts of the currently selected OM class or object symbol. If a DP node or component symbol is selected the stereotype and properties texts can be edited. For a component symbol, integration model can also be selected. For a DP thread symbol, the thread priority, stack size, queue size and max signal size texts are editable. For a DP object symbol the stereotype, properties and qualifier texts can be edited.

The Symbol Details window and its appearance are described in "Symbol Details Window" on page 1701 (for OM symbols) and in "Symbol Details Window" on page 1714 (for DP symbols).

Class

This **OM** command brings up the modal *Browse & Edit Class* dialog on the currently selected OM class or object. This dialog is used to inspect and edit OM classes and objects over page and diagram boundaries.

The Browse & Edit Class dialog is described in detail "Browse & Edit Class Dialog" on page 1691.

Drawing Size

Issues a dialog where the width and height can be adjusted. The values will be saved.

Enlarging the drawing, the current page may not fit any longer into the paper that is defined with the Print preferences; the result may be a page that requires multiple sheets of paper when printed.

Select All

This operation selects all objects contained within the drawing area, or all text in the text window.

Status

This **MSC** command displays and possibly modifies the *status* of timers.

For more information see "Displaying and Modifying Status" on page 1721.

Make Space

This **MSC** command presents the *Make Space* dialog, where space for events may be inserted.

- The radio buttons *Before selected object* and *After selected object* define whether space for insertion of events should be created **before** or **after** the currently selected object.
- The *Space of* text field allows to specify how many events should be possible to insert.
- Clicking *OK* inserts space according to the number of events. All objects which are found after the currently selected object are pushed downwards.

Redirect

This **OM** command changes the orientation of the selected aggregation or generalization line.

This **SC** command changes the orientation of the selected SC transition line(s).

This **DP** command changes the orientation of the selected DP association line, if the line currently has a direction. The direction of a DP association line can be changed in the Line Details window.

This **MSC** command changes the orientation of a selected *message*. The new direction is indicated by a change in the orientation and position of the *message end* and of the *message parameters*.

Connect

This **MSC** command opens a dialog that is used to connect a selected *condition* or *MSC reference* symbol to one or multiple instances.

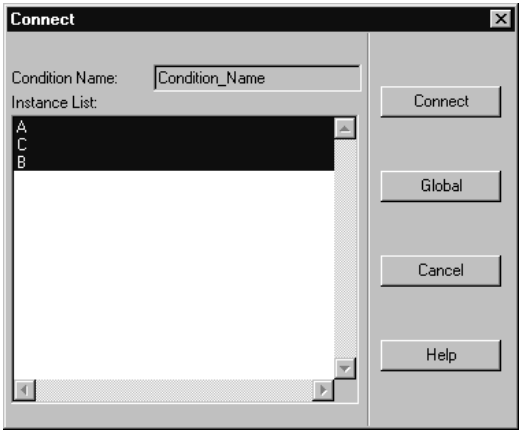


Figure 298: The Connect dialog (for a condition)

The *Name* field allows you to enter or change the name of the condition or MSC reference.

The *Instance List* lists all possible instances that are concurrent with the condition or MSC reference.

- Instances in the list that are selected or already connected are highlighted. Clicking an entry in the list toggles its state between highlighted and not highlighted.
- Select *Connect* connects the condition or MSC reference symbol to the instances in the instance list that are currently selected.
- Selecting *Global* connects the condition or MSC reference symbol globally to all instances in the list.

Decompose

This **MSC** command creates a new diagram of an instance axis in the original diagram. The instance axis you decompose is marked decomposed. In the new diagram you can specify the interior of the decomposed instance.

The messages to and from the decomposed instance are displayed as found and lost messages (see "MSC Symbols and Lines" on page 1651), showing the communication with other instance axes. The decomposed diagram is editable and it is displayed in the Organizer.

View Menu

The *View* menu provides rescaling functions and access to various options that affect the behavior of the editor. The *View* menu contains the following menu choices:

- *Set Scale*
- *Temporary Colors > Remove*
- *Window Options*
- *Diagram Options*
- *Editor Options*
- *Insert Options*

Set Scale

This menu choice issues a dialog where you can adjust the scale.

Temporary Colors > Remove

Temporary colors are colors that are not saved in the diagram. These colors are used by certain operations, such as *Compare Diagrams* in the MSC or HMSC editor, to highlight symbols of interest. Before doing a new operation that relies on color, it might be useful to remove already existing temporary colors. This can either be done by using this menu choice or by closing down the editor.

Window Options

This menu choice issues a dialog where you can set the options that affect the window properties.

In the dialog, you can set if you want the following items to be displayed:

- *Tool Bar*
- *Status Bar*
- *Symbol Menu*

  On UNIX, the space allocated to the symbol menu will be reused by the drawing area when you hide the symbol menu.

- *Instance Ruler*

  The space allocated to the **MSC** instance ruler will be reused by the drawing area when you hide the instance ruler.

- *Text Window*

  On UNIX, the space allocated to the text window will be reused by the drawing area when you hide the text window.

- *Page Breaks*

  This option determines whether physical page breaks, with the appearance of dashed horizontal and vertical lines, should be displayed or not in the drawing area. These page breaks are defined by the print preferences. Also, a number is shown in the bottom of each printout page, indicating the physical page number when printed.

- *Show Grid*

Click *OK* to apply the options in the dialog to the current window only.

Click *All Windows* to apply the options in the dialog to all windows opened by the editor.

Diagram Options

This **MSC** command issues a dialog where you can set the options that determines what should be displayed in the drawing area:

- *Show Instance Name*
- *Show Instance Kind*
- *Show Instance Composition*
- *Show Message Name*
- *Show Message Parameters*

Click *OK* to apply the options to the current MSC. The diagram options will be saved when you save the file.

Editor Options

Opens a dialog where you can customize the behavior of the editor.

The options are controlled by toggle buttons. They are:

- *Always new Window*

  This option indicates whether or not a new window should be opened when a new diagram is opened or created. The default behavior is not to open a new window.

- *Sound*

  This option indicates whether or not improper actions in the editor, such as attempting to append a symbol to an illegal position, should be brought to your attention by producing an alert sound. The default value for this option is on.

- *Show Link Endpoints*

  This option indicates whether endpoint markers should be displayed for symbols having link endpoints. The default value is on.

Insert Options

This **MSC** command issues a dialog where spacing between symbols may be specified. (The dimmed parts of the dialog are for future use.)

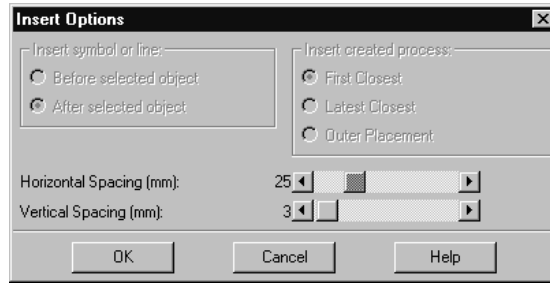


Figure 299: The Insert Options dialog

Spacing

This option allows to modify the space which is automatically inserted between the symbols and lines when appended to the chart.

There are two scales for setting the minimum horizontal distance between *instance axes* and the minimum vertical distance between *messages*.

Pages Menu

This section applies to all editors except MSC and DP.

The *Pages* menu holds commands that assist you in navigating among the pages in a diagram that are currently being edited in the editor. It also contains commands for adding, renaming and clearing pages as well as clipboard functions for copying, cutting and pasting entire pages.

The *Pages* menu contains the following menu choices:

- *First*
- *<Page Name>*
- *Last*
- *Add*
- *Edit*

First

This menu choice opens the first page contained in the diagram. The first page is defined according to the order of appearance in the *Edit Page* dialog.

<Page Name>

Activating the *Pages* menu presents up to four menu choices that consist of the names of the two pages that are sequentially immediately before and after the page being edited. If the first page of the diagram is being edited, the next four sequential pages are shown. If the last page of the diagram is edited, the previous four pages are shown.

Selecting one of these page names opens or restores that page in the editor. To open other pages, the *Edit* menu choice is used.

Last

This menu choice opens the last page of a diagram.

Add

This menu choice is a shortcut for adding one page to the current diagram. The *Add Page* dialog is issued and after pressing the *OK* button the new page is shown.

Edit

This menu choice allows to *Add*, *Rename*, *Move up*, *Move down*, *Clear*, *Cut*, *Copy* and *Paste* a page. Following this menu choice, a dialog is issued.

The meaning of the various components is as follows:

Edit Pages List

Presents a list with all pages that are included in the diagram. Clicking on a page in this list selects it and makes it the subject of the operation to follow.

Edit Button

Clicking this button opens the selected page and displays it in an editor window. The *Edit Page* dialog is closed.

Cut

Clicking this button removes the selected page from the diagram and saves it in the clipboard buffer.

Copy

Clicking this button copies the selected page into the clipboard buffer.

Paste

Clicking this button pastes the page contained in the clipboard buffer into the current diagram. A new dialog is issued.

The dialog allows to specify:

- The new name of the page. By default the page is autonumbered (the preference is described in "AutoNumberPages" on page 280 in chapter 3, *The Preference Manager*).
- If the new page should be pasted before or after the current page.

- Clicking the *Paste* button pastes the page according to the options as set up in the dialog and returns control to the *Edit Page* dialog.

Clear

This operation clears (removes) the selected page from the diagram.

To confirm the operation, click on *Clear*. The editor will automatically rename autonumbered pages.

> **Caution!**
>
> As stated in the dialog, *Clear* on a page cannot be undone.

Move up

If a diagram page is selected and you click the *Move up* button, the page will be moved up.

> **Note:**
>
> The undo function in the Organizer and in the SDL Suite will have no effect on page moves. It is, however, easy to undo a page move operation by just moving the page back again.

Move down

If a diagram page is selected and you click the *Move down* button, the page will be moved down.

> **Note:**
>
> The undo function in the Organizer and in the SDL Suite will have no effect on page moves. It is, however, easy to undo a page move operation by just moving the page back again.

Add

This command creates a new page which is added to the current diagram.

A new dialog is issued, in which you must insert the page name. You can insert the new page either before or after the current page.

The dialog allows to specify:

- The page name. By default the page is autonumbered (this preference is described in "AutoNumberPages" on page 280 in chapter 3, *The Preference Manager*).
- *Before / After current page*
  The new page is inserted according to the status of these radio buttons.
- Clicking *OK* adds the page and returns control to the *Edit Page* dialog.

Rename

Click this button to open a dialog where you can rename the selected page.

> **Note:**
> Autonumbered pages cannot be renamed. The *autonumbered* option must first be turned off.

- Type in the new name of the page.
- Click *OK* to rename the page. Control is returned to the *Edit Page* dialog.

Autonumbered

When turning the *Autonumbered* option off, the editor first prompts to confirm the removal of autonumbering on that page.

- Click *Yes* to remove autonumbering on the page and then open a dialog where you can rename the page (see "Rename" on page 1681).
- Click *No* to cancel the operation. The page remains autonumbered.

Turning the *Autonumbered* option on applies a numeric name to the selected page (1, 2, etc...). A dialog is issued where:

- *Yes* auto-numbers the selected page.
- *All Pages* auto-numbers all pages contained in the diagram.
- *No* closes the dialog without autonumbering the pages.

Open this page first

This toggle button designates what page to be opened first when opening a diagram in the editor, if no particular page is specified.

Diagrams Menu

The *Diagrams* menu records all diagrams that are opened by the editor. The menu choices are:

- *Back*
- *Forward*
- *<Diagram Name>*
- *List All*

Back

Select this menu choice to browse back to the diagram that was previously displayed in the window.

Forward

Select this menu choice to browse forward to the diagram that was displayed in the window before you selected *Back*.

<Diagram Name>

The last edited diagram always goes to the top of the list, and subsequently moves the other diagrams down a position. A maximum of 9 open diagrams can be shown. A tenth one will be put at the top of the list, but any subsequent opening of a diagram will only show the last 9 that have been opened.

Each item in the menu provides information about the diagram type, its name, a slash (‘/’) followed by a page name, a hyphen and, possibly, the file it is stored on (the file information is missing if the diagram has never been saved).

A diagram that is preceded by an asterisk (‘*’) denotes that it has been modified during the editor session.

> **Note:**
>
> OM, SC, DP and HMSC diagrams are listed in the same *Diagrams* menu, whereas MSC diagrams are listed only in the *Diagrams* menu of the MSC Editor window.

List All

This menu choice becomes available when a maximum of 9 open diagrams has been surpassed. When *List All* is selected, it provides a dialog containing all diagrams that are currently open in the editor. Select a diagram and click *Edit* to display it.

Window Menu

The *Window* menu contains the following menu choices:

- *New Window*
- *Close Window*
- *Entity Dictionary*
- *Info Window*
- *<Window Name>*
- *List All*

New Window

This command opens a new editor window containing a new view on the page, MSC or DP diagram contained in the source window from which this menu choice was operated. The page, MSC or DP diagram can be edited in any window.

Close Window

This option closes the open window, **but**, not necessarily the diagram.

If more than one editor window is opened, only the current window is closed and not the diagram. If the last open editor window is closed, the editor will act as if *Exit* has been chosen, possibly in conjunction with a save of information.

For more information, see "Close Diagram" on page 14 in chapter 1, *User Interface and Basic Operations*).

Entity Dictionary

Opens the Entity Dictionary window. See "The Entity Dictionary" on page 434 in chapter 9, *Implinks and Endpoints*.

Info Window

This **MSC** command issues a dialog as shown in Figure 300, displaying additional information about the currently selected MSC object.

Figure 300: The Info dialog

The dialog presents:

- A scrollable information window providing information about the currently selected object. What information is available depends on the selected object. See "Displaying Information About the Selected MSC Object" on page 1722.
- *Show SDL Symbol*

  A button that allows, **if the MSC has been automatically generated, using the SDL Simulator or SDL Explorer**, to obtain a trace-back to the SDL source diagrams. Manually inserted symbols and lines do not, of course, contain any trace-back information. Clicking this button displays the SDL source symbol in an SDL Editor window. To behave properly, this feature requires that the SDL source diagram is consistent with the generated MSC, e.g. the SDL diagram may not have been modified so that the source symbol has been removed.

<Window Name>

If more than one editor window is open the other windows are listed here. The behavior of this list will be similar with the diagrams list in the "Diagrams Menu" on page 1682. The only difference is that the menu items will not provide the diagram file information.

List All

This menu choice will be available only if more than 9 editor windows are open, and have the same functionality as the *List All* menu choice in the *Diagrams* menu.

Tools Menu

The *Tools* menu contains the following menu choices:

- *Show Organizer*
- *Link > Create*
- *Link > Create Endpoint*
- *Link > Traverse*
- *Link > Link Manager*
- *Link > Clear*
- *Link > Clear Endpoint*
- *Search*
- *Spelling > Comments*
- *Spelling > All Text*
- *Connect to Text Editor*
- *Show GR Reference*
- *Create Bookmark*
- *Convert SC to SDL*
- *Filter*
- *Tidy Up*
- *Navigate*
- *Rearrange*
- *Generate MSC PR*
- *Generate Input Script*

All Link commands are described in "Link Commands in the Tools Menus" on page 442 in chapter 9, *Implinks and Endpoints*.

Search

This menu choice allows you to search for a text in the current diagram or document in an editor. The search can be restricted to a selected symbol type by using the *Search In* option menu.

You can search several diagrams and documents at the same time by using search from the Organizer, see "Search" on page 142 in chapter 2, *The Organizer*.

Spelling > Comments

Check the spelling of comments in the current diagram. For more information about the spelling operation, see "Spelling > Comments" on page 147 in chapter 2, *The Organizer*.

Spelling > All Text

Check the spelling of all text in the current diagram. For more information about the spelling operation, see "Spelling > Comments" on page 147 in chapter 2, *The Organizer*.

Connect to Text Editor

This command issues an external text editor and creates a temporary file from the currently selected text. The text can from now on only be edited in the external editor. The editor is updated every time the external text editor saves the temporary file. When the temporary file is no longer edited, the editing control returns to the editor.

Which external text editor to use is defined by the preference SDT*TextEditor.

Show GR Reference

This command issues a message where the graphical reference for the currently selected object is displayed.

The syntax of the graphical references used in the editor is described in chapter 18, *SDT References*.

Create Bookmark

This command creates a bookmark in the Organizer for the object you have currently selected.

Convert SC to SDL

This **SC** command transforms a State Chart to an SDL process diagram.

When the conversion is done, the SDL Editor displays the newly created process diagram. A new diagram is created for every conversion. The diagram resides in an unsaved SDL Editor buffer.

The transformation rules that are applied to the SC diagram are described in "Converting State Charts to SDL" on page 1702.

Filter

This SC command allows you to hide uninteresting states, transitions or signals in a statechart. All three filters are combined, making it possible to hide stateX, transition (stateA)-signalB->(stateC) and signalZ at the same time.

Information about hidden entities is saved in a collapsed text symbol, to be able to recreate the entities when the filter is removed. The text in the text symbol starts with “HiddenTransitions:”.

The dialog has the following options:

Apply filter

The current filter will be used when the dialog is closed with the OK button.

Create new editor buffer

A new statechart is created as a new editor buffer. If this option is off, the current statechart and editor buffer will be reused instead.

Always tidy up afterwards

The Tools>Tidy Up operation will be invoked after the filter has been applied. If this option is off, then the Tidy Up operation will only be invoked when needed, i.e. when states or transitions that was not visible before the operation become visible.

Create filter script text symbol

When the dialog is closed with the OK button, another dialog will appear, asking for a name for the filter. A text symbol, containing information about the current filter, will be created, starting with the text “Filter:<newline><filter name>”. To reuse a filter saved in this way, select the text symbol before invoking the filter operation.

Tidy Up

This **SC** command rearranges the graphical layout of states and transitions in the State Chart. State hierarchies are flattened out, i.e. states in states are replaced with other constructs. Positions and sizes are determined by a default layout algorithm.

Navigate

The *Navigate* menu choice displays a (H)MSC diagram referenced by the reference symbol or a decomposed MSC.

Since a reference symbol may contain more than one reference name the HMSC Editor presents a dialog where you can choose reference name.

Goto

Pressing this button will show the diagram selected in the Navigation names list. If an (H)MSC diagram with that name is not present in the Organizer, a dialog will be shown. If more than one (H)MSC with the same name is present in the Organizer, an error message will be issued.

There are a number of situations when the dialog is not presented and the referenced diagram is shown immediately:

1. There is only one reference name in the symbol.
2. The caret is positioned in a reference name. This name will be automatically selected for navigation.

For reference navigation to work correctly it is required that the expression in the symbol is syntactically correct so that the names in the symbol can be extracted. Check out "Syntax rules in Symbols" on page 1740. If you try to navigate from a symbol with incorrect syntax you will be notified with a dialog that points out the location of the error in the reference symbol text.

If you select a symbol that is not present in the Organizer you will be presented with a dialog where you can create either an MSC or an HMSC diagram.

Rearrange

This **MSC** command rearranges the symbols and lines in the MSC diagram to get a compact version of the same diagram. This command might for instance be useful after having removed several signals from the diagram.

Generate MSC PR

With this **MSC and HMSC** command you can save an MSC using the Z.120 MSC/PR format. (MSC/PR can also be read by the MSC Editor,

see "Managing MSCs" on page 1797 in chapter 41, *Editing MSC Diagrams*). The MSC Editor normally stores the MSCs using a binary storage format.

To generate MSC/PR:

1. Display the MSC you want to generate MSC/PR from.
2. Select *Generate MSC PR* from the *Tools* menu. A dialog is issued.
3. Specify a file where to store the generated MSC/PR. The default file that the MSC Editor suggests consists of the name of the MSC, with the `.mpr` file extension, i.e. `<diagramname>.mpr`
4. Make sure the *Generate GR references*[1] radio button is turned on.
5. Click *Generate*.

Comment Symbols and MSC/PR

Since free comment symbols, i.e. comment symbols that are not connected to any other symbol, are not permitted in MSC/PR but can be defined using the MSC Editor, free comment symbols are converted to text symbols in the generated MSC/PR.

Generate Input Script

This **MSC** command makes it possible to convert an MSC expressed in a certain, restricted way to an input script that can be used as a script in the SDL Simulator UI. In the Simulator UI, use the Execute>Input Script menu choice or the execute-input-script command to execute an input script.

How MSCs should be expressed to be able to use them as test scripts for the SDL Simulator is described in "Simulator Test > New Simulator" on page 156 in chapter 2, *The Organizer*.

Note that normally, you do not manually convert MSCs to input scripts using this menu choice. Instead, the Organizer Tools menu choice Simulator Test > New Simulator auto-converts MSCs to input scripts when they are needed as test cases. Manual conversion is only needed if you

1. Event oriented MSC/PR describes the MSC using the order in which the events occur, i.e. starting with the top of the diagram and downwards, providing the feeling of a global event order.

want to stop working on the MSC level and start working on the textual Simulator UI command level instead.

Compare Diagrams

With this menu choice you can compare the contents of two MSC or HMSC diagrams. See "Comparing and Merging Diagrams" on page 1745.

Merge Diagrams

With this menu choice you can compare the contents of two MSC or HMSC diagrams and create a new diagram by merging the two diagrams. See "Comparing and Merging Diagrams" on page 1745.

File Menu of the Text Window

The *File* menu provides functions that transfer text from a file to the text window and vice-versa. The basic intention is to provide you a means to edit larger portions of text with a more suitable text editor. Another possibility to edit text externally is to use the *Connect to Text Editor* command in the *Tools* menu. The available menu choices are:

- *Import*
- *Export*

Import

Import imports the contents of a file into the text window and inserts the contents of the file at the current I-beam cursor position, possibly replacing selected text in the text window. A file selection dialog is issued, where the file to import text from is to be specified. The file name filter is set to `*.txt` by default.

Export

Export exports the text window contents to a file. A file selection dialog is issued, where the file to export the text to is to be specified. The file name filter is set to `*.txt` by default.

OM Editor Specific Information

Browse & Edit Class Dialog

The Browse & Edit Class dialog is opened when you select *Class* from the *Edit* menu. The dialog allows inspection of OM diagram classes and objects across page and diagram borders.

An object model class is defined as the union of the attributes and operations in the class and object symbols, see "Class Definition Summary" on page 1731, and the purpose of the Browse & Edit Class dialog is to display and ensure consistency when editing this combined information.

The Browse & Edit Class dialog is a modal dialog which is divided in two parts. The browsing functionality is placed at the top part of the dialog, where all classes in the scope, and all occurrences of each class are listed in two option menus. Below, in the editing part, the name of the class, all attributes and all operations are available.

The Browse & Edit Class dialog is available when a single class or object symbol has been selected:

- If a class symbol is selected, the Browse & Edit Class dialog will operate on the class defined by that symbol. If the class name is empty or contains incorrect syntax, the Browse & Edit Class dialog will not be available.
- If an object symbol is selected, the Browse & Edit Class dialog will operate on the class that the object symbol instantiates. If the object symbol does not include the name of the class it instantiates, the Browse & Edit Class dialog will not be available.

Browse

By using the topmost part of the Browse & Edit Class dialog it is possible to browse amongst all class and object symbols within the scope. The scope is either the entire Organizer module where the diagram is contained, or the diagram itself if it is not contained in an Organizer module. The module concept is described in "Module" on page 43 and in "Module" on page 52 in chapter 2, *The Organizer*.

Edit

By using the lower part of the Browse & Edit Class dialog it is possible to edit the name, attributes and operations of a class. The change will propagate into all class symbols of this class in OM diagrams within the module where the diagram is contained. The attributes and operations will be presented in a list. This list is parsed information from the texts in the class symbols. If a particular text contains syntax errors, it may lead to that attributes and operations contained in the same text do not appear in this list.

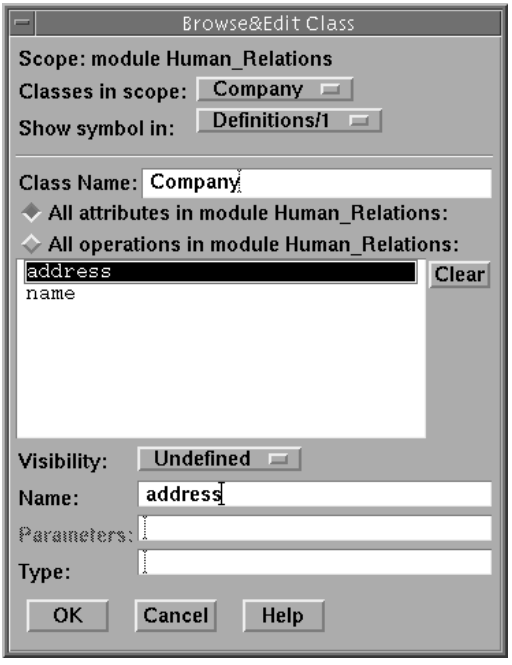


Figure 301: The Browse & Edit Class dialog

The *Scope* Label

The scope label is a non-editable text field that describes the scope that the Browse & Edit Class dialog is operating in. The scope can be either a single OM diagram or all OM diagrams within an Organizer module.

- If the diagram containing the object from which the Browse & Edit Class dialog was invoked is not part of a module in the Organizer, the scope label contains the text “Scope: diagram <*diagram name*>”.
- If the diagram containing the object from which the Browse & Edit Class dialog was invoked is part of a module in the Organizer, the scope label contains the text “Scope: module <*module name*>”.

The *Classes in Scope* Option Menu

The *Classes in Scope* option menu contains a list of all classes within the scope. By selecting one of the classes in this menu the corresponding symbol will be shown and selected in the drawing area.

The *Show Symbol in* Option Menu

One class can be represented by several class symbols. The *Show Symbol in* option menu lists all occurrences of the class currently selected in the Classes in Scope option menu. The notation in the menu is <diagram-name>/<page-name>. By selecting an occurrence in this menu the corresponding symbol will be shown and selected in the drawing area.

The *Class Name* Field

The name field is an editable text field that initially contains the name of the class that is being edited.

By editing the class name field, it is possible to update all occurrences of that class name in class and object symbols in the current scope when the *OK* button is clicked.

The *Attributes/Operations* Buttons

The Attributes and Operations buttons select whether the attributes/operations list will contain a list of all attributes or all operations defined for the selected class. Only one of the attributes and operations buttons will be selected at any time.

The *Attribute/Operations* List

The attributes/operations list contains an alphabetically sorted list of the attributes or operations defined for the currently selected class.

The *Clear* Button

The clear button removes the currently selected operation or attribute from the *Attribute/Operations* list.

By clearing an attribute or operation using the clear button, that attribute or operation will be removed from all relevant class and object symbols using the current class name in the current scope when the *OK* button is clicked.

The *Visibility* Option Menu

The visibility option menu contains the visibility of the currently selected attribute or operation in the *Attribute/Operations* list, if any.

By selecting one of the predefined values in this menu and later clicking the *OK* button, the visibility of the selected attribute or operation will be updated in all relevant class symbols using the current class name in the current scope.

The *Name* Field

The name field contains the name of the currently selected attribute or operation in the *Attribute/Operations* list, if any.

Editing this field will change the definition of that attribute or operation in all relevant class and object symbols using the current class name in the current scope when the *OK* button is clicked.

The *Parameters* Field

The parameters field is an editable text field that contains the parameters of a class operation. It is only editable when an operation is selected in the *Attribute/Operations* list.

Editing this field will change the definition of the selected operation in all relevant class and object symbols using the current class name in the current scope when the *OK* button is clicked.

The *Type* Field

The type field is an editable text field that is only editable when an attribute or an operation is selected in the *Attribute/Operations* list.

Depending on the current selection, the type field contains:

- If an attribute is selected, the type of that attribute, if any.
- If an operation is selected, the return type of that operation.

By editing this field, the type of the selected attribute or the return type of the selected operation and later clicking the *OK* button, the type of the selected attribute or operation will be updated in all relevant class and object symbols using the current class name in the current scope.

Note that default values for attributes cannot be inspected or changed in the Browse & Edit Class dialog.

The *OK* Button

The *OK* button will close the Browse & Edit Class dialog and update all appropriate class and object symbols in the diagrams in the current scope as described for the scope label.

Note that no changes are made in the diagrams until the *OK* button is clicked. This makes it possible to specify several changes in the dialog and later disregard them by clicking the *Cancel* button.

The values are syntactically checked, so it is not possible to add syntax errors to your classes by using the Browse & Edit Class dialog. Also, it may be impossible to delete syntactically incorrect text from symbols using the Browse & Edit Class dialog, since the Browse & Edit Class dialog relies on information obtained by parsing the relevant symbol text compartments.

It is possible to undo the changes made by the Browse & Edit Class operation. Note that this undo operation may affect more than one diagram.

The *Cancel* Button

The *Cancel* button will close the Browse & Edit Class dialog and discard any changes specified in the dialog. However, if the selection has changed in the drawing area after using any of the browsing functionality, this selection will not be canceled.

All changes made in the Browse & Edit Class dialog will be lost.

Line Details Window

The Line Details window is opened when you select *Line Details* from the *Edit* menu. It is used to inspect, and edit the properties of the currently selected line. In particular, most of the line attribute objects that are available for the different line types can only be created from the Line Details window[1].

Some line attribute objects contain editable text, and clearing that text, whether by editing in the Line Details window or directly in the diagram, will remove the attribute.

Changes made in the Line Details window will take immediate effect and will be shown in the drawing area. These changes can be undone with the *Undo* menu command.

Unlike the Browse & Edit Class dialog, the Line Details window is modeless and can remain open while you continue to work with the diagrams. The contents of the window will be updated to reflect the current selection. If there is none or more than one selected line, the fields in the Line Details window will be dimmed.

The OM Editor supports four different types of lines and the contents of the Line Details window depends on the type of the currently selected line:

- If an *Association* line is selected, all items except the Composite button in the Line Details window will be active. See Figure 302.
- If an *Aggregation* line is selected, all items in the Line Details window will be active. See Figure 303.
- If a *Generalization* line is selected, only a single editable text field, containing the generalization's discriminator, will be available. See Figure 304.
- If a *Link Class* line is selected, all items in the Line Details window will be dimmed.
- If none or more than one line is selected, all items in the Line Details window will be dimmed.

1. Once created, however, all editable textual line attribute objects can be selected and edited directly in the diagram, without the use of the Line Details window.

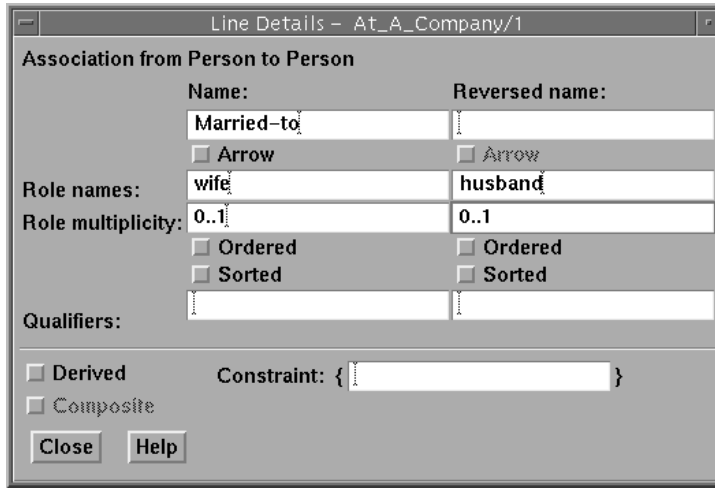


Figure 302: The Line Details window when an association line is selected

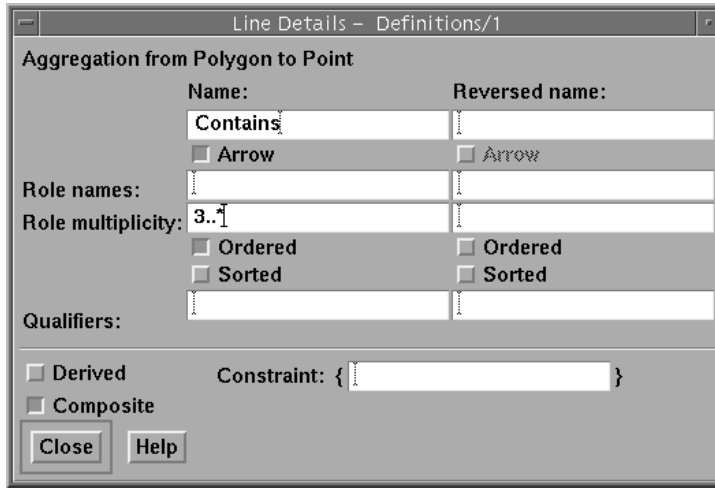


Figure 303: The Line Details window when an aggregation line is selected

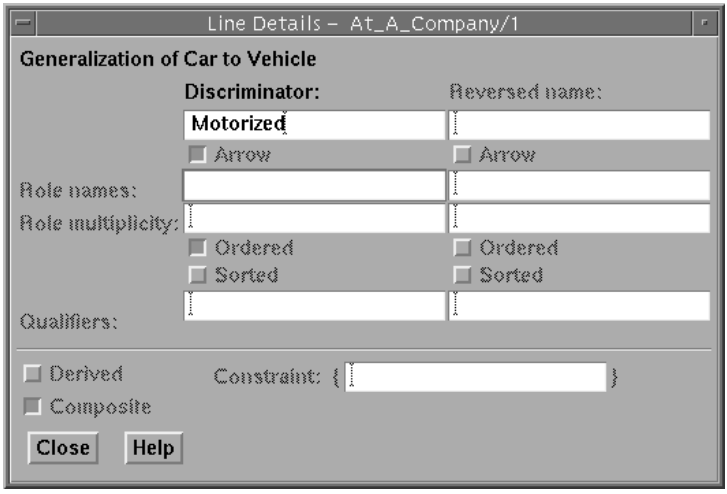


Figure 304: The Line Details window when a generalization line is selected

The *Name* Field

The *Name* field is an editable text field that contains the name of the selected aggregation or association.

Unlike the other text fields in the Line Details window, the line attribute object defined by the name field is permanent and is not destroyed even if the name field is cleared.

This field is only available when an aggregation or association line is selected.

The *Discriminator* Field

The *Discriminator* field is an editable text field that changes the discriminator text attribute of the generalization.

This field is only available when a generalization line is selected.

The *Reversed Name* Field

The *Reversed Name* field is an editable text field that allows the specification of a reversed name for an aggregation or association.

This field is only available when an aggregation or association line is selected.

The *Arrow* Buttons

Each of these two buttons toggle an option that shows or hides the optional arrow defining the direction of the name attribute or the reversed name attribute, respectively.

The direction and position of the arrow will be automatically calculated from the position and size of the associated name attribute as well as the direction of the line.

Typically it is desirable to use arrows to enhance clarity when defining both name and reversed name fields.

These buttons are only available when an aggregation or association line is selected.

The *Role Name* Fields

The *Role Name* fields are editable text fields that allow you to create and edit the role attributes belonging to each end of the association or aggregation.

These fields are only available when an aggregation or association line is selected.

The *Role Multiplicity* Fields

The *Role Multiplicity* fields are editable text fields that allow you to create and edit the multiplicity attributes of each end of an association or aggregation.

The name field is only available when an aggregation or association line is selected.

The *Ordered* Buttons

Selecting the *Ordered* toggle creates an uneditable text attribute containing the text "{ordered}". Deselecting the *Ordered* toggle removes this text attribute.

The ordered text attribute can be specified both in the primary and the reverse direction.

These buttons are only available when an aggregation or association line is selected.

The *Sorted* Buttons

Selecting the *Sorted* toggle creates an uneditable text attribute containing the text "{sorted}". Deselecting the *Sorted* toggle removes this text attribute.

The sorted text attribute can be specified both in the primary and the reverse direction.

These buttons are only available when an aggregation or association line is selected.

The *Qualifiers* Fields

The *Qualifiers* fields are editable text fields that allow you to create and edit the qualifier fields in the forward and reverse direction respectively.

These fields are only available when an aggregation or association line is selected.

The *Derived* Button

The *Derived* button is a toggle option that indicates whether the selected association or aggregation should be marked as derived, i.e. crossed by a small slanting line.

This button is only available when an association line is selected.

The *Composite* Button

The *Composite* button is a toggle option that indicates whether the selected aggregation should be marked as composite, i.e. the diamond shape of the aggregation line should be filled.

This button is only available when an aggregation line is selected.

The *Constraint* Field

The *Constraint* field is an editable text field that allows you to create and edit the constraint line attribute object.

This field is only available when an association or aggregation line is selected.

The *Close* Button

The *Close* button closes the Line Details window.

Symbol Details Window

The Symbol Details window is opened when you select *Symbol Details* from the *Edit* menu. It is used to inspect, and edit the stereotype and properties fields of the currently selected class or object symbol. These two symbol attribute types can only be created from the Symbol Details window. However, when created they can be edited directly in the diagram.

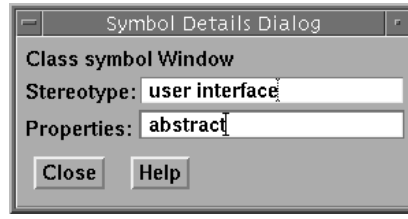


Figure 305: The Symbol Details dialog

Changes made in the Symbol Details window will take immediate effect and will be shown in the drawing area. These changes can be undone with the *Undo* menu command.

Unlike the Browse & Edit Class dialog, the Symbol Details window is modeless and can remain open while you continue to work with the diagrams. The contents of the window will be updated to reflect the current selection. If multiple class or object symbols or any other symbol or line is selected the Symbol Details window will be dimmed.

The placement of the stereotype and properties texts in the class and object symbol is described in "Class Symbols" on page 1638.

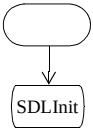
SC Editor Specific Information

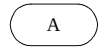
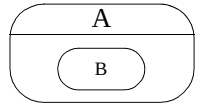
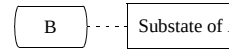
Converting State Charts to SDL

To convert state charts to SDL, you select *Convert SC to SDL* from the *Tools* Menu.

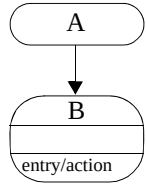
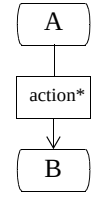
Following are the transformation rules that are applied to the SC diagram:

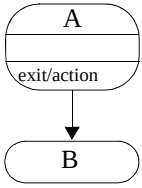
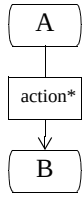
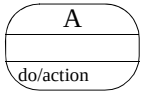
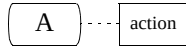
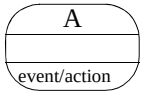
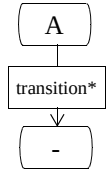
Transformation Rules for Diagrams and Symbols

SC	SDL
SC diagram	Process diagram
“Heading”	Process “Heading”
Text symbol	Text symbol
Start symbol  Note: This rule does not apply if the start symbol belongs to a hierarchical state.	Start symbol 
If an event is defined on the transition from this state or if more than one transition from this state:	Start symbol connected to ‘initial’ state symbol named SDLInit. 

SC	SDL
Termination symbol  Note: 1. This rule does not apply if the termination symbol belongs to a hierarchical state. 2. If more than one transition to the termination symbol, the stop symbol is duplicated.	Stop symbol 
State symbol 	State symbol 
Hierarchical state symbol	Nothing
Substate symbol 	State symbol with a comment symbol 

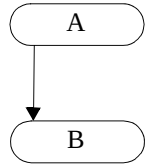
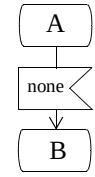
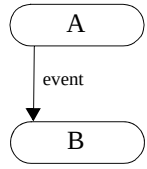
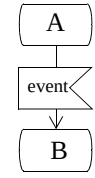
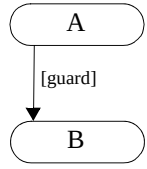
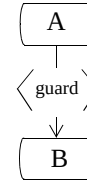
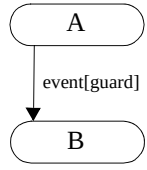
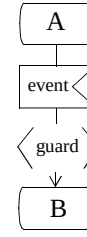
Transformation Rules for State Internal Activities

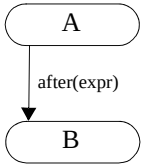
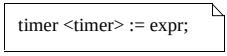
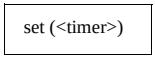
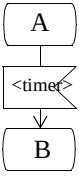
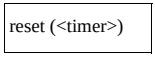
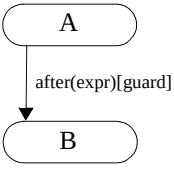
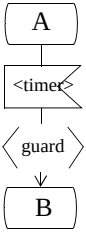
SC	SDL
For all transitions from A to B 	Add action symbol  * See “Transformation Rules for Actions” on page 1710

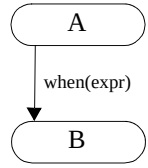
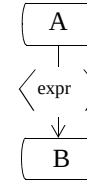
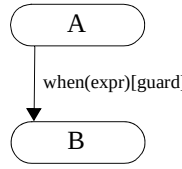
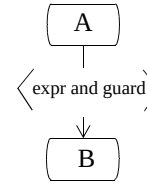
SC	SDL
For all transitions from A to B 	Add action symbol  * See "Transformation Rules for Actions" on page 1710
	Comment symbol 
	 * See "Transformation Rules for Transitions" on page 1704

Transformation Rules for Transitions

If actions are specified in the transition labels, insert actions last in the generated transition.

SC	SDL
	Spontaneous transition 
	
	Continuous signal  Note: Priority is required if more than one continuous signal exist from the same state. This must be manually added.
	Enabling condition 

SC	SDL
	Timer with the name "<A>_T<counter>", where <counter> is an integer.  For all transitions to <A> add:  For this transition from <A>:  For all other transitions from <A> add: 
	As above, but add an enabling condition: 

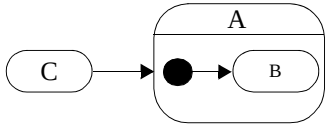
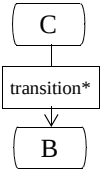
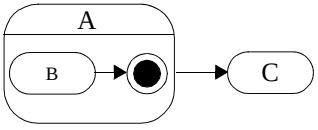
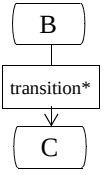
SC	SDL
	Continuous signal  Note: Priority is required if more than one continuous signal exist from the same state. This must be manually added.
	Continuous signal  Note: Priority is required if more than one continuous signal exist from the same state. This must be manually added.

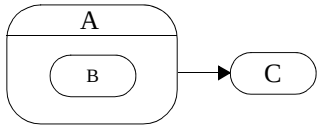
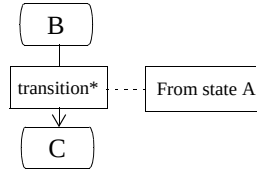
Transformation Rules for State Internal Activities on Hierarchical States

- An **Entry** action of a hierarchical state is always performed when one of its substates is entered from the outside. This is applied for any number of state boundaries crossed.
- An **Exit** action of a hierarchical state is always performed when one of its substates takes a transition to a state outside the substate region. This is applied for any number of state boundaries crossed.

- An **Event** action of a hierarchical state is applied to all of its sub-states, at any nesting depth, unless a transition with the same event exists on the substate.
- A **Do** action of a hierarchical state is applied to all of its substates, at any nesting depth.

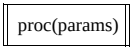
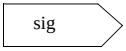
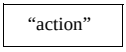
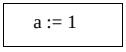
Transformation Rules for Transitions on Hierarchical States

SC	SDL
 Note: Exactly one start state, with an unlabeled transition, should exist in the hierarchical state.	 * See "Transformation Rules for Transitions" on page 1704
 Note: An unlabeled transition should exist out from the hierarchical state.	 * See "Transformation Rules for Transitions" on page 1704

SC	SDL
 Note: 1. This rule applies for every substate at any nesting depth. 2. Exception: This rule does not apply if a transition with the same trigger (event[guard]) exists on the substate.	 * See “Transformation Rules for Transitions” on page 1704

Transformation Rules for Actions

Depending of text contents:

SC	SDL
If text starts with ‘call’: *call proc(params)*	Procedure call 
If text starts with ‘output’: *output sig*	Output 
If quoted text: *“action”*	Informal task 
Otherwise: *a := 1*	Task 

DP Editor Specific Information

Line Details Window

The Line Details window is opened when you select *Line Details* from the *Edit* menu. It is used to inspect, and edit the properties of the currently selected line. In particular, most of the line attribute objects that are available for the different line types can only be created from the Line Details window[1].

Some line attribute objects contain editable text, and clearing that text, whether by editing in the Line Details window or directly in the diagram, will remove the attribute.

Changes made in the Line Details window will take immediate effect and will be shown in the drawing area. These changes can be undone with the *Undo* menu command.

The Line Details window is modeless and can remain open while you continue to work with the diagrams. The contents of the window will be updated to reflect the current selection.

The DP Editor supports two different types of lines and the contents of the Line Details window depends on the type of the currently selected line:

- If an *Association* line is selected, the text fields for the name, protocol and encoding as well as the direction option menu in the Line Details window will be active. See Figure 306.
- If an *Composite Aggregation* line is selected, only the multiplicity text field in the Line Details window will be active. See Figure 307.
- If none or more than one line is selected, all items in the Line Details window will be dimmed.

1. Once created, however, all editable textual line attribute objects can be selected and edited directly in the diagram, without the use of the Line Details window.

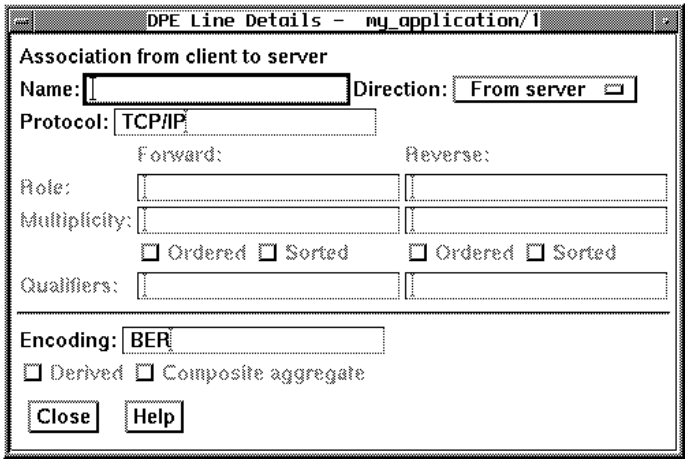


Figure 306: The Line Details window when an association line is selected

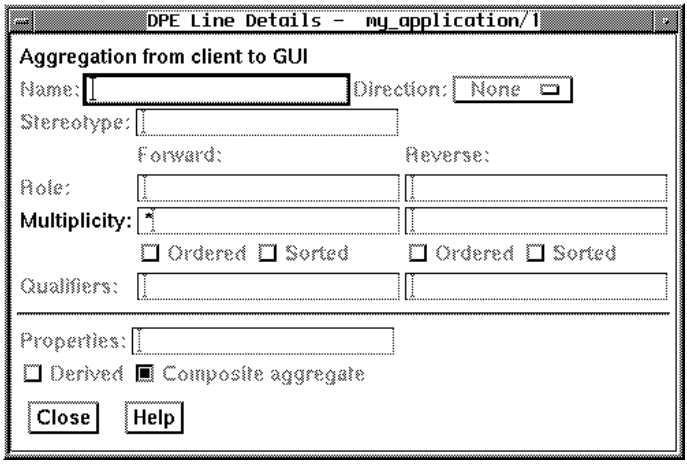


Figure 307: The Line Details window when an aggregation line is selected

The *Name* Field

The *Name* field is an editable text field that contains the name of the selected association.

This field is only available when an association line is selected.

The *Direction* Option Menu

By selecting one of the predefined values in the direction option menu the direction attribute of the currently selected association can be set. If any other value than None is selected an arrow will appear at one of the ends of the association line.

This option menu is only available when an association line is selected.

The *Protocol* Field

The *Protocol* field is an editable text field that allows you to create and edit the protocol attribute of an association.

The protocol field is only available when an association line is selected.

For an association line the line attribute object defined by the protocol field is permanent and is not destroyed even if the protocol field is cleared. In this respect it differs from most of the other text fields in the Line Details window.

The *Encoding* Field

The *Encoding* field is an editable text field that allows you to create and edit the encoding attribute of an association.

The encoding field is only available when an association line is selected.

The *Multiplicity* Field

The *Multiplicity* field is an editable text field that allows you to create and edit the multiplicity attribute of an aggregation.

The multiplicity field is only available when an aggregation line is selected.

For an aggregation line the line attribute object defined by the multiplicity field is permanent and is not destroyed even if the multiplicity field is cleared. In this respect it differs from most of the other text fields in the Line Details window.

The *Composite aggregate* Button

The *Composite aggregate* button is a toggle option that indicates that the selected aggregation is a composite aggregation, i.e. the diamond shape of the aggregation line is filled.

This button is set automatically depending on the current selection and is therefore never available.

The *Close* Button

The *Close* button closes the Line Details window.

Symbol Details Window

The Symbol Details window is opened when you select *Symbol Details* from the *Edit* menu. It is used to inspect, and edit the stereotype and properties fields of the currently selected node, component, thread or object symbol. For the component symbol the integration model can be selected. For the thread symbol you can edit thread priority, stack size, queue size and max signal size. For the object symbol you can enter a qualifier. These symbol attribute types can only be created from the Symbol Details window. However, when created the stereotype and properties texts can be edited directly in the diagram. All other settings can only be inspected and edited in the Symbol Details window.

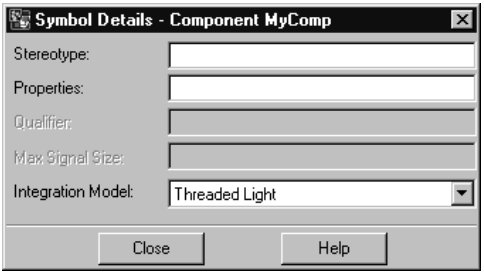


Figure 308: The Symbol Details dialog

Changes made in the Symbol Details window will take immediate effect and will be shown in the drawing area. These changes can be undone with the *Undo* menu command.

The Symbol Details window is modeless and can remain open while you continue to work with the diagrams. The contents of the window will be updated to reflect the current selection. If multiple symbols or a line is selected the Symbol Details window will be dimmed.

Generating a Partitioning Diagram Model

The Partitioning Diagram Model is used by the Targeting Expert as an hierarchical overview of the entities that are to be built. Partitioning Diagram Models can be generated from Deployment Diagrams. More information about this can be found in chapter 40, *The Deployment Editor, in the User´s Manual*. That chapter contains guidelines for modeling Deployment Diagrams that can be used for targeting, as well as information on the available integration models.

You generate Partitioning Diagram Models from the Organizer by selecting a Deployment diagram and then selecting “Targeting Expert” in the pop-up menu. You can also select “Targeting Expert” from the Generate menu. If the generation is successful the Targeting Expert is launched with the Partitioning Diagram Model as input.

Mapping Rules

Associations are ignored in Partitioning Diagram Model generation.

The stereotype “external” on a component or node symbol means that the symbol and its sub-tree are not included when the Partitioning Diagram Model is generated. This makes it possible to model the environment that surrounds an SDL system in a deployment diagram.

Graphical Syntax Rules

In order to generate a valid Partitioning Diagram Model that can be used by the Targeting Expert the rules listed below have to be followed:

- A valid diagram contains at least one node, one component and one object, which are connected.
- at least one component must belong to each node
- at least one thread or one object must belong to each component
- at least one object must belong to each thread
- every symbol must have a name
- names of nodes must be unique within the diagram
- names of components must be unique within their node

- names of threads must be unique within their component
- each thread must be connected to exactly one component
- each object must be connect to one component or one thread

Object symbols

For object symbols several text attributes have to be correct in order to result in valid Partitioning Diagram Models.

The name of each object symbol in the diagram should be the name of the corresponding SDL entity. Whether it is a system, block or process should be specified by the stereotype and the qualifier text should describe how the SDL entity fits into the tree structure of the SDL system.

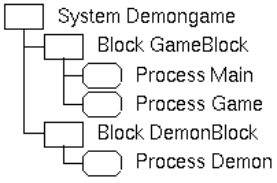


Figure 309: The tree structure of the system Demongame

Figure 309 shows the system Demongame. An object representing the process Main in a deployment diagram should have the following text attributes:

- qualifier: **`Demongame/GameBlock/Main`**
- stereotype: **`process`**
- name: **`Main`**

In the same way an object representing the system Demongame should have **`Demongame`** as qualifier text, the stereotype **`system`** and the name **`Demongame`**.

MSC Editor Specific Information

Pasting in MSC Diagrams

To paste, you select *Paste* from the *Edit Menu*. The rules for *Paste* will be described below.

When pasting the selection, the MSC Editor will process each individual object contained in the selection, adjust it to the grids if required and connect it, if feasible, to the “closest” object(s) in the drawing area.

If the MSC Editor fails in pasting some of the objects contained in the selection, the rest of the objects will nevertheless be pasted but without an error message being displayed.

Pasting of Multiple Objects

When pasting a selection that consists of multiple objects, the MSC Editor attempts, as long as feasible, to preserve the original appearance of the selection.

Pasting of Individual Objects

When pasting individual objects, the MSC Editor processes the objects as follows:

- *Text*

  A text symbol may be pasted anywhere.

- *Comment*

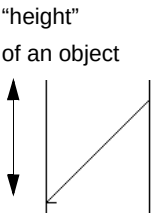
  A comment symbol may be pasted anywhere. The line connecting the comment symbol to another symbol is however lost when pasting.

- *Instance head* / *Instance axis*

  The instance head and its axis may be pasted anywhere.

- *Instance end* and *Stop*

  An instance end or stop may only be pasted at a location where it overlaps an instance axis.



- *Message*

  A message is connected to the instance axes which are closest to the message's base and end point, preserving the direction and "height" of the message.

- *Message-to-self*

  A message-to-self is connected to the closest instance axis. The "height" of the message is preserved.

- *Condition* and *MSC Reference*

  A condition or an MSC reference is connected to the instance it was previously connected to (before copying it to the clipboard).

- *Timer*

  A timer is connected to the closest instance axis. The "height" of the timer is preserved (see *Message* above).

- *Action*

  An action is pasted on the closest instance axis.

- *Create*

  The create symbol may be pasted anywhere, but only together with the instance it is creating.

- *Coregion*

  A coregion is connected to the closest instance axis, to the left or right. The "height" of the coregion is preserved (see *Message* above).

Adding Symbols

Adding an Instance Head Symbol

When you add an instance head symbol, an instance axis line is drawn that connects the symbol to the bottom of the drawing area. The instance axis line cannot be drawn manually. It will be truncated or elongated when you add or move an instance end or stop symbol upwards or downwards. See "Adding an Instance End Symbol" on page 1807 in chapter 41, *Editing MSC Diagrams* and "Adding a Stop Symbol" on page 1808 in chapter 41, *Editing MSC Diagrams*.

The instance head symbol has three text fields:

- The instance name field
- The instance kind field
- The decomposition field. See Figure 310.

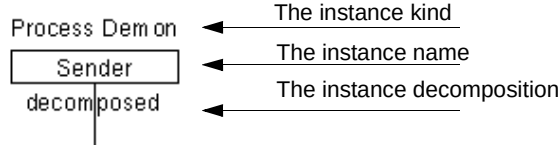


Figure 310: The meaning of the instance head text fields

The ITU recommendation Z.120 discusses several ways of using the instance name and instance kind text fields. In the MSC Editor, the text fields are used as follows:

- Basically, the *instance kind* reflects what kind of instance is represented. That is, an identifier which reflects the name of the corresponding SDL block/service/process/procedure; and an optional identifier which identifies if it is a block, a service or a process (the *kind denominator*).
- The *instance name* reflects the current function of the instance.
- The *instance decomposition* shows if the instance is in turn decomposed as a sub MSC. This text field is by default empty, but can contain the text *decomposed* or *decomposed as <diagram name>* to indicate that the instance is in turn decomposed. To decompose an instance, you select *Decompose* from the *Edit* menu.

Adding a Condition, MSC Reference or Inline Expression Symbol

A *condition* or *MSC reference* describes either a global system state referring to all instances contained in the MSC or a state referring to a subset of instances (a non-global condition or MSC reference). The minimum subset is a single instance.

For two MSCs, the second is a continuation of the first if its initial global condition is identical to the final global condition of the first. Identifying an intermediate global condition can be used when breaking down an MSC into two parts.

By means of non-global conditions, combinations of MSCs with different sets of instances can also be defined. The continuation then refers only to the common subset of instances.

The following general rule applies to the continuation of two MSCs (MSC1 and MSC2, with a non-empty common set of instances).

MSC2 is a continuation of MSC1 if, for each instance which both MSCs have in common:

- MSC1 ends with a (non)global condition
- MSC2 begins with a corresponding[1] (non)global condition
- Each (non)global condition of MSC2 has a corresponding (non)global condition in MSC1.

An *inline expression* symbol is always global and connected to all instances. The inline expression symbol is created from two inline expression symbol parts:

- The inline expression symbol starts a new inline expression symbol. The inline expression text in this symbol determines the inline expression type:
  - The keyword **loop** identifies a loop. Example: “LOOP <1, 5>” means that the contents of the inline expression symbol will be executed between one and five times.
  - The keyword **opt** identifies an optional part. The contents of the inline expression symbol will either not be executed or executed once.
  - The keyword **exc** identifies an exceptional part. The contents of the inline expression symbol will either not be executed or executed once. If it is executed, then the rest of the MSC is skipped.
  - The keyword **par** identifies parts that are executed in parallel.
  - The keyword **alt** identifies alternative parts. Only one part will be executed.
- The inline expression separator symbol starts a new part in a parallel or alternative inline expression.

1. *Corresponding* in this context means that both conditions refer to the same subset of instances and both conditions agree with respect to name.

Displaying and Modifying Status

To display and modify status, you select *Status* from the *Edit* menu.

Timer Status

Z.120 specifies two appearances for the timer symbol, depending on whether the timer has expired (i.e. time-out) or whether the timer is reset. The appearance of the timer is illustrated in Figure 311.

- **Timeout** – A timer expires.
- **Reset** – The timer is stopped.
- **Implicit reset** – Assigned a new value while active.

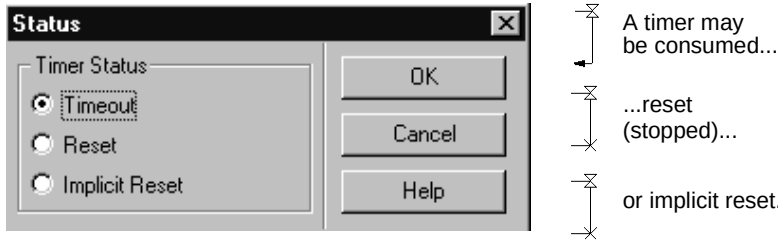


Figure 311: Timer status

Separate Timer Status

Separate timer symbols has no need for an implicit reset. The timer can be set two times in a row without an implicit reset in-between.

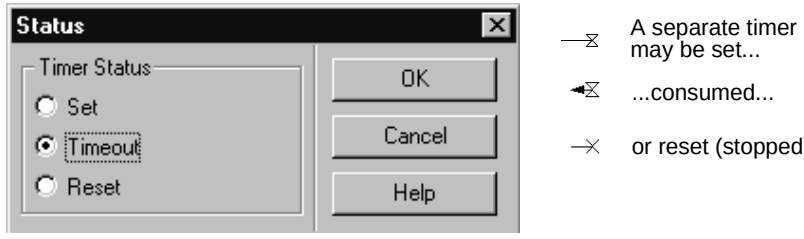


Figure 312: Separate Timer status

Displaying Information About the Selected MSC Object

To display information about a selected MSC object, you select *Info Window* from the *Window* menu.

This opens a dialog displaying additional information about the currently selected MSC object. The information that is available depends on what object is selected, and this is described in the table below.

Selected Object	Information Available
Instance head	• Instance name • Instance kind • Composition • Creating instance
Instance end	• Instance kind
Stop	• Instance kind
Process create	• Creating instance • Created instance • When the instance was created[a] • SDL Reference[a]
Message output (the message output is selected by clicking close to the message base)	• Message name • Sending instance • Receiving instance • Message status = – Sent[b] – Consumed[c] • When the message was sent (now)[a] • Message parameters • SDL Reference[a]

Selected Object	Information Available
Message input (the message input is selected by clicking close to the message end)	• Message name • Sending instance • Receiving instance • Message status= – Sent[b] – Consumed[c] • When the message was consumed • Message parameters • SDL Reference[a]
Timer	• Timer name • Instance kind • Timer status= – Reset (stopped) – Implicit reset – Timeout • When the message was sent (now)[a] • Timer value (Set) • Timer parameters • SDL Reference[a]
Separate timer	• Timer name • Instance kind • Timer status= – Set – Reset (stopped) – Timeout • When the message was consumed (now)[a] • Timer parameters • SDL Reference[a]
Action symbol	• Instance kind • Action text • SDL Reference[a]
Condition	• Condition or MSC reference name • Instances connected to the condition • SDL Reference[a]

Selected Object	Information Available
MSC Reference symbol	• Condition or MSC reference name • Instances connected to the condition
Text symbol	• Symbol text
Comment symbol	• Symbol text
Inline symbol	• Instances connected • Operator (inline expression)

a. Only for diagrams created through a simulation.

b. A message is *sent* when it has been sent and received, but has not yet been *consumed* by the receiving instance. Typically, it is waiting in the receiving process input queue.

c. A message is *consumed* when it has been processed by the receiving instance.

Instance Ruler

When managing working on a long (vertically extended) MSC, the *kind* of instance in the *instance head* may not be visible in the drawing area.

The *instance ruler* (illustrated in Figure 313) shows the kind of the instance heads that are not currently visible in the drawing area. It reduces the amount of scrolling up or down when working on an MSC. The instance ruler may be shown or hidden with an option (see "Instance Ruler" on page 1676).

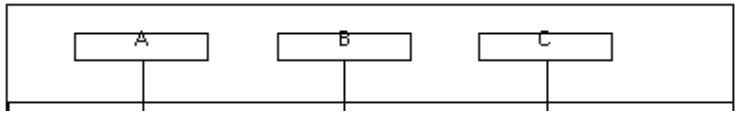


Figure 313: The instance ruler

Tracing a Simulation in a Message Sequence Chart

This section describes the functionality behind tracing a simulation in an MSC.

General

The MSC Editor may be used as a graphical trace tool, which features the automatic generation of an MSC from a simulation.

- The results of the simulation may be presented on line in an MSC Editor window, in which each event of interest will be appended to the chart in order to build up an MSC which reflects the history of the simulation.
- It is also possible to run the simulation, save the results on a simulator MSC log file and open the file from the MSC Editor.
- The commands that start up the logging of MSC events, set up the scope of trace, stop the logging of events, and so on, are given to the simulator monitor. See chapter 50, *Simulating a System*.

Defining the Scope of Trace

First, the scope of trace should, if necessary, be set to the unit which is currently of interest. This is done with the command Set-MSC-Trace.

The graphical trace is then started with a dedicated command from the simulator monitor. See *"Start-Interactive-MSC-Log" on page 2178 in chapter 49, The SDL Simulator*. In response to this command, the following happens:

1. In the Organizer, a reference to an MSC is added. The diagram is assigned an unique name.
2. An instance of the MSC Editor window is activated on the newly created MSC.
3. The current status for the simulation is presented by displaying all SDL process instances that exist at the time when ordering the command Start-Interactive-MSC-Log.

Mapping Between SDL and MSC

The mapping rules which govern how SDL events are transformed into MSC symbols, lines and textual elements are summarized in the following table:

SDL concept	MSC Concept
Signal • Output • Input	Message • Sending • Consumption
Signal to self • Output • Input	Message to self • Sending • Consumption
Timer • Set • Reset • Implicit Reset • Input	Timer • Set • Reset • Implicit reset[a] • Time-out
Create a process	Process Create
Stop a process	Stop
Environment	Environment[b]
System <system name>	Instance <instance kind>
Substructure <system name>	Instance <instance kind>
Block <block name>	Instance <instance kind>
Process <process name> <instance number>	Instance <instance kind> <instance name>
State	Condition[c]
Task	Action[d]
Comment	Comment

a. See "Implicit reset" on page 1729.

b. The system's environment is denoted by env_1

c. The mapping is only completely valid for a condition connected to only one instance. A condition connected to several instances expresses a logical AND-combination of the states of the processes corresponding to those instances.

d. The *action* symbol corresponds to a task symbol containing informal text. This translation is, however, not supported when generating an MSC from a simulation.

Generating the MSC

When running the simulation, each SDL event of interest (in the scope of MSC Trace) that takes place will cause the corresponding MSC symbol to be drawn in the drawing area of the MSC Editor.

Drawing Conventions

Layout

Default layout algorithms are used. Each event causes the insertion point to be translated downwards with one vertical spacing unit, keeping the intuitive feeling of **absolute order** between events. An event could be, for instance, the output or the input of an SDL signal. However, if an output event is immediately followed by an input event, no translation will take place.

Auto-Resizing of MSC

The MSC Editor will automatically enlarge the MSC size when the MSC has grown so that it reaches the bottom or right of the drawing area. The MSC is enlarged according to a preference parameter.

Messages

A distinction is made between the *reception* and the *consumption* of messages:

- **Reception** of a message: Once an SDL signal is output in the simulator, it is immediately placed into the input port of the receiving process instance. There may, however, be other pending SDL signals in this queue, which means that the signal might not be input immediately. Therefore, when a signal is output, it is illustrated as a

lost message, marked at its end with the name of the receiving instance, from the *sending instance* to the *receiving instance*.

- **Consumption** of a message: When an SDL signal is input, the vertical position for the next event may have moved down in comparison to the position of the signal output. An input of a signal is illustrated by redrawing the line representing the message with its end point connected to the new vertical position. (If an output event is immediately followed by an input, there is no change in vertical position.)
- In the case messages remain lost for a “long” time, this may be interpreted as some kind of erroneous behavior which requires special attention. Messages that are never consumed indicate that some design error might have been introduced in the SDL system. It is up to you to decide whether the time which has elapsed since a message was sent should be considered as exceeding a reasonable value.

Create of Process

Each time an SDL process is dynamically created, a process create will be drawn from the source instance axis, and the created instance will be positioned according to the insert and grouping modes that is set. This guarantees that no overlapping instance axes will take place.

The MSC Editor will automatically enlarge the MSC size when the MSC has grown so that it reaches the bottom or right of the drawing area. The drawing area is enlarged according to the value of a preference parameter.

Process Stop

If an instance is stopped (by a stop symbol), the space which becomes available beneath the instance end symbol will not be reused for new instance axes.

Timers

- The MSC Editor can handle four different timer statuses resulting from a simulation:

  – *Set*
  – *Reset (stopped)*
  – *Implicit reset*
  – *Timeout*

  On the MSC they are shown according the figure below:

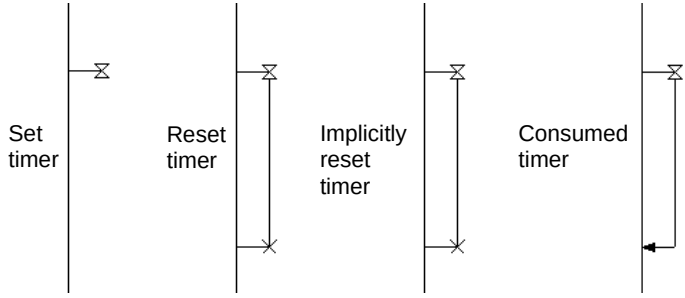


Figure 314: Timer status as shown on the MSC

  – *Implicit reset*

    The *implicit reset* timer status is a non-Z.120 addition to the MSC Editor. An implicitly reset timer is immediately set again to its original parameters after reset. It is illustrated in the same way as a *reset* timer, but the status information is reflected in the *Info* dialog (see "Requesting Detailed Information on an Object" on page 1802).

Instances

- **The Environment Instance**

  All interchange of information between the SDL system and its environment is displayed by sending / receiving messages to or from an instance with the name `env_1`. This instance is normally placed at the left of the drawing area.

- **The Void Instance**

  Graphically, in the Message Chart Editor, the concept *conditional trace* (see *"Scope of Trace for Generation of Message Sequence Charts" on page 2187 in chapter 49, The SDL Simulator*) is illustrated as sending or receiving messages to / from an *instance* with the name *Void*. The purpose of the *Void* instance is to document that a message exchange actually took place without focusing on the sending / receiving instance.

- **The Instance Name and Instance Kind Text Fields**

  The ITU recommendation Z.120 allows several ways of using the two text fields *Instance Name* and *Instance Kind*. This was discussed earlier in "Adding an Instance Head Symbol" on page 1806 in chapter 41, *Editing MSC Diagrams*.

  When generating an MSC, the SDL *Process Name* and the internal *instance number* managed by the simulator monitor are concatenated to build up one (MSC) *Instance Name*. The SDL *diagram type* and *diagram name* are concatenated to build one *Instance Kind*.

  Example 315 Instance Name and Instance Kind in Simulation Trace

  Assume the simulation program contains an instance of the SDL diagram type = *process* with the *diagram name* = Demon and monitor *instance number* = 2 This would be displayed as the following instance:

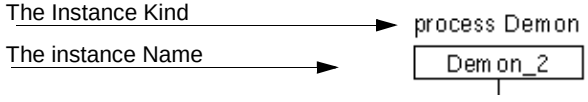


Figure 315: The use of the instance text fields in a simulation

Terminating the Trace

Having terminated the trace (either by stopping the simulation or turning the MSC trace off), **you are responsible for saving** the MSC(s) that are the results of the simulation.

Syntax Summary

Object Model Syntax

While the intent of Object Model diagrams is to give as large flexibility as possible to analysis and specification activities, the different editing support facilities provided by the OM Editor requires textual data to follow certain syntactic conventions.

This section details the syntax rules that are imposed by the OM Editor on textual data only. Graphical syntax rules are enforced automatically by the OM Editor and are not described here.

Summary of Lexical Rules

In general, a somewhat restricted version of the lexical rules of SDL apply for names. These rules have the following noteworthy properties:

- It is possible to break a name over more than one line by using the underscore at the end of the line to escape the newline.
- It is possible to have identifiers consisting entirely of numeric characters; e.g. “223” is a valid identifier.
- Identifiers are not case sensitive.
- Comments can be used anywhere in the texts. A comment is defined as an SDL comment, i.e. `/* this is a comment */`

For more information on the SDL lexical rules, see the Z.100 recommendation.

The primary restriction is that identifiers containing spaces are not considered legal.

```
<name> ::= identifier as described above.
<c-string> ::= string as bounded by " characters.
<sdl-string> ::= string delimited by '
(apostrophes).
```

Class Definition Summary

When interpreting OM diagrams, all class and object symbols within all pages within all diagrams in a single module must be considered. A class definition is considered to be the conceptual union of the informa-

tion contained in all class symbols with the same name, as well as all object symbols declared as instances of that class.

This means that the definition of a class can be distributed over several symbols, pages and even diagrams. The definitions in class symbols with the same name are then merged to obtain the complete definition of the class; support for this is provided by the *Browse & Edit Class* dialog.

Thus, when defining classes, it is acceptable to have identical attributes or operations defined several times, either in the same or different class or object symbols. The identical definitions are simply redundant, and do not affect the correctness of the complete class definition.

Note, however, that each unnamed class objects is considered distinct, so that each unnamed class symbol defines its own class.

Diagram Name Syntax

Diagram names must follow certain restrictions since they are used to identify the diagram with respect to other tools.

```
Diagramname ::= <name>
```

Page Name Syntax

```
Pagename ::= <name>
```

Each page name must be unique within its containing diagram.

Class Symbol Syntax

The textual information in class symbols is divided into five compartments of which three have their own syntax. The stereotype and properties compartments are not checked for syntax.

In the attributes and operation compartments the *stereotype concept* as described in "Object Model Literature References" on page 1767 can be used. The syntax chosen supports the single characters left/right guillemet («») but it can also be written as two angle brackets. To type the left/right guillemet in the text window, press `<Ctrl>` when you type the left/right angle bracket.

Class Name Compartment Syntax

```
Classname ::= <name> |
   <name> ':' ':' <name> |
   <empty>
```

Class Attributes Compartment Syntax

```
ClassAttributesCompartment ::=
   { <stereotype> <classattribute> <note> }*
<stereotype> ::= '<' '<' <name> '>' '>' |
   '«' <name> '»' |
   <empty>
<classattribute> ::=
   <visibility> <attribute>
<visibility> ::=
   '+' |
   '-' |
   '#' |
   <empty>
<attribute> ::=
   <name> <value> |
   <name> ':' <type> <value>
<type> ::= <name>
<value> ::= '=' <valueitem> | <empty>

<valueitem> ::=
   <name> |
   '-' <name> |
   '+' <name> |
   <c-string> |
   <sdl-string> |
   '(' <valueitemlist> ')' |
   '{' <valueitemlist> '}'
<valueitemlist> ::=
   <valueitem> |
   <valueitemlist> ',' <valueitem>
<note> ::=
   '{' <name> '}' |
   '{' <name> '=' <name> '}' |
   <empty>
```

Two attributes are considered identical if they have the same name and type. Note that attributes defining different values are still considered identical.

Class Operations Compartment Syntax

```
ClassOperationsCompartment ::=
   { <stereotype> <classoperation> <note> }*
<classoperation> ::=
   <visibility> <operation>
<operation> ::=
   <name> <return> |
   <name> '(' ')' <return> |
   <name> '(' <parameterlist> ')' <return>
```

```
<parameterlist> ::=
   parameter |
   parameterlist ',' parameter
<parameter> ::=
   <type> |
   <name> ':' <type>
<return> ::=
   ':' <name> |
<empty>
```

Two operations are considered identical only if the name, parameter list and the return fields are all the same. Thus “op”, “op(a)” and “op:t” are all unequal, while “op(a)”, “OP(A)” define the same operation.

Note however that “op” and “op()” are not considered equal, since the OM Editor does not interpret an omitted parameter list as an empty parameter list.

Object Symbol Syntax

The textual information in object symbols is divided into four compartments of which two have their own syntax. The stereotype and properties compartments are not checked for syntax.

Object Name Compartment Syntax

```
Objectname ::=
   <name> ':' <classname> |
   <name> |
   <empty>
```

Object Attributes Compartment Syntax

```
ObjectAttributesCompartment ::=
   { <stereotype> <attribute> <note> }*
```

Text Symbol Syntax

Text symbols are not subject to any syntax rules.

Line Syntax

The current version of the OM Editor does not implement any syntax checks on the contents of the textual attribute objects of lines.

State Chart Syntax

While the intent of State Chart diagrams is to give as large flexibility as possible to analysis and specification activities, the different editing support facilities provided by the SC Editor requires textual data to follow certain syntactic conventions.

This section details the syntax rules that are imposed by the SC Editor on textual data only. Graphical syntax rules are enforced automatically by the SC Editor and are not described here.

Summary of Lexical Rules

In general, a somewhat restricted version of the lexical rules of SDL apply for names. These rules have the following noteworthy properties:

- It is possible to break a name over more than one line by using the underscore at the end of the line to escape the new line.
- It is possible to have identifiers consisting entirely of numeric characters; e.g. “223” is a valid identifier.
- Identifiers are not case sensitive.
- Comments can be used anywhere in the texts. A comment is defined as an SDL comment, i.e. `/* this is a comment */`

For more information on the SDL lexical rules, see the Z.100 recommendation.

The primary restriction is that identifiers containing spaces are not considered legal.

```
<name> ::= identifier as described above.
```

State Definition Summary

UML allows some expressions in text segments to be defined by the tool vendors. For State Charts such expressions should follow SDL/PR syntax, but this is not checked by the SC Editor.

When interpreting SC diagrams, all state symbols within all pages within all diagrams in a single module must be considered. A state definition is considered to be the conceptual union of the information contained in all state symbols with the same name.

This means that the definition of a state can be distributed over several symbols, pages and even diagrams. The definitions in state symbols with the same name are then merged to obtain the complete definition of the state.

Thus, when defining states, it is acceptable to have identical activities defined several times, either in the same or different state symbols. The identical definitions are simply redundant, and do not affect the correctness of the complete state definition.

Note, however, that each unnamed state object is considered distinct, so that each unnamed state symbol defines its own state.

Diagram Name Syntax

The diagram name identifies the diagram.

```
<DiagramName> ::= <name>
```

Page Name Syntax

The page name identifies the page within a diagram.

```
<PageName> ::= <name>
```

Each page name must be unique within its containing diagram.

State Symbol Syntax

The textual information in state symbols' state sections is divided into three compartments with different syntax.

Name Compartment Syntax

The name identifies the state.

```
<NameCompartment> ::=
     [<name>]
```

Internal Activity Compartment Syntax

Internal activities' syntax is defined by the following grammar:

```
<InternalActivityCompartment> ::=
     {<internal-activity>}*

<internal-activity> ::=
     <event-signature>
     [<guard-condition>]
```

```
      ['/' <action-expr>]
      |
      <reserved-action-label> '/' <action-expr>

<reserved-action-label> ::=
      'entry' | 'exit' | 'do'

<event-signature> ::=
     <name> [<argument-list>] |
     <predefined-event> '(' <argument> ')'

<predefined-event> ::=
      'when' | 'after'

<argument-list> ::=
     '(' [<argument> {',' <argument>}*] ')'

<argument> ::=
      <expression>

<guard-condition> ::=
      '[' <expression> ']'

<action-expr> ::=
      <expression>

<expression> ::=
      SDL/PR expression
```

Two events are considered identical only if the name and the argument list are all the same. Thus “op” and “op(a)” are unequal, while “op(a)”, “OP(A)” define the same event.

Note however that “op” and “op()” are not considered equal, since the SC Editor does not interpret an omitted parameter list as an empty parameter list.

Text Symbol Syntax

Text symbols are not subject to any syntax rules.

Transition Line Syntax

The textual information in the *transition label* attribute of transition lines is defined by the following grammar:

```
<TransitionLabel> ::=
      [<event-signature>]
      [<guard-condition>]
      [<action-list>]
```

```
<action-list> ::=
     '/' <action-expr> {';' <action-expr>}*
```

DP Syntax

While the intent of Deployment diagrams is to give as large flexibility as possible to analysis and specification activities, the different editing support facilities provided by the DP Editor require textual data to follow certain syntactic conventions.

This section details the syntax rules that are imposed by the DP Editor on textual data only. Graphical syntax rules are enforced automatically by the DP Editor and are not described here.

Summary of Lexical Rules

In general, a somewhat restricted version of the lexical rules of SDL apply for names. These rules have the following noteworthy properties:

- It is possible to break a name over more than one line by using the underscore at the end of the line to escape the newline.
- It is possible to have identifiers consisting entirely of numeric characters; e.g. “223” is a valid identifier.
- Identifiers are not case sensitive.
- Comments can be used anywhere in the texts. A comment is defined as an SDL comment, i.e. `/* this is a comment */`

For more information on the SDL lexical rules, see the Z.100 recommendation.

The primary restriction is that identifiers containing spaces are not considered legal.

```
<name> ::= identifier as described above.
```

Diagram Name Syntax

The diagram name identifies the diagram.

```
<DiagramName> ::= <name>
```

Node Symbol Syntax

The name identifies the node.

```
<NodeName> ::= <name> |
                    <empty>
```

There are no syntax checks on the stereotype and properties texts.

Component Symbol Syntax

The name identifies the component.

```
<ComponentName> ::= <name> |
                    <empty>
```

There are no syntax checks on the stereotype and properties texts.

Thread Symbol Syntax

The name identifies the thread.

```
<ThreadName> ::= <name> |
                    <empty>
```

Object Symbol Syntax

The name identifies the object.

```
<ObjectName> ::= <name> |
                    <empty>
```

There are no syntax checks on the stereotype, properties and qualifier texts.

Text Symbol Syntax

Text symbols are not subject to any syntax rules.

Line Syntax

The DP Editor does not implement any syntax checks on the contents of the textual attribute objects of lines.

HMSC Syntax

While the intent of High level MSC diagrams is to give as large flexibility as possible to analysis and specification activities, the different editing support facilities provided by the HMSC Editor requires textual data to follow certain syntactic conventions as specified in MSC'96.

This section details the syntax rules that are imposed by the HMSC Editor on textual data only. Graphical syntax rules[1] are enforced automatically by the HMSC Editor and are not described here.

Summary of Lexical Rules

The lexical elements follow MSC'96. These rules have the following noteworthy properties:

- An underscore followed by on or more spaces is ignored completely, e.g. A_ B denotes the same <name> as AB. This use of underline makes it possible to split keywords over multiple lines since control characters (e.g. newline) is treated as a space.
- It is possible to have identifiers consisting entirely of numeric characters; e.g. "223" is a valid identifier.
- Identifiers are not case sensitive.
- Comments (`<note>` in Z.120) can be used anywhere between lexical units. A `<note>` is started by the characters `/*` outside a `<note>` and terminated by the characters `*/` inside a `<note>`.

For more information on the MSC lexical rules, see the Z.120 recommendation.

Syntax rules in Symbols

The descriptions below describes what text is allowed in the different symbols, it is not a description of the grammar in PR-form.

The grammar below informally describes the lexical units used as terminal symbols. Refer to Z.120 for a complete description.

```
<name> ::= identifier as described above.
<text> ::= approximately the ASCII character set.
```

1. No checks are made on completeness, however.

In the sections below each symbol will be described in the following manner:

1. A *short* description of the symbol and/or its contents and purpose.
2. The symbols textual grammar
3. Possibly any semantic restriction and/or notes. Refer to Z.120 for a complete description of the semantics that apply.

Diagram Name Syntax

The diagram name identifies the MSC to the outside world.

```
Diagramname ::= <name>
```

Diagram names must follow certain restrictions since they are used to identify the diagram with respect to other tools.

Page Name Syntax

The page name identifies the page within a document.

```
<page name> ::= <name>
```

Each page name must be unique within its containing diagram.

Text Symbol Syntax

The text in a text symbol is an informal explanatory text.

```
<text text> ::= <text>
```

The text has no (formal) semantic and no restrictions.

Condition Symbol Syntax

Global conditions can be used to restrict how MSC's can be composed in High level MSC's.

```
<condition name list> ::=
   <condition name> { ',' <condition name> }*

<condition name> ::= name
```

Global conditions indicate possible continuations of Message sequence charts containing the same the same set of instances by means of condition name identification.

Reference Symbol Syntax

MSC References are used to refer to other MSC's of the MSC document.

```
<msc ref expr> ::=
   <msc ref par expr> { 'alt' <msc ref par expr> }*

<msc ref par expr> ::=
   <msc ref seq expr> { 'par' <msc ref seq expr> }*

<msc ref seq expr> ::=
   <msc ref exc expr> { 'seq' <msc ref exc expr> }*

<msc ref exc expr> ::=
   [ 'exc' ] <msc ref opt expr>

<msc ref opt expr> ::=
   [ 'opt' ] <msc ref loop expr>

<msc ref loop expr> ::=
   [ 'loop' [ <loop boundary> ] ]
   {
     'empty' |
     <msc name> [ <parameter substitution> ] |
     ( <msc ref expr> )
   }

<parameter substitution> ::=
   'subst' <substitution list> 'end'

<substitution list> ::=
   <substitution> [ ';' <substitution list> ]

<substitution> ::=
   <replace message> |
   <replace instance> |
   <replace msc>

<replace message> ::=
   [ 'msg' ] <message name> 'by' <message name>

<replace instance> ::=
   [ 'inst' ] <instance name> 'by' <instance name>

<replace msc> ::=
   [ 'msc' ]
   { 'empty' | <msc name> } 'by'
   { 'empty' | <msc name> }

<msc name> ::= <name>

<message name> ::= <name>

<instance name> ::= <name>
```

MSC Syntax

While MSC diagrams allows the user to specify the analysis and specification activities in great detail with graphical symbols, there are also fields that get their meaning from textual information only. The different editing support facilities provided by the MSC Editor checks textual data to follow certain syntactic conventions as specified in MSC'96.

This section details the syntax rules that are imposed by the MSC Editor on textual data only. Graphical syntax rules[1] are enforced automatically by the MSC Editor and are not described here.

Summary of Lexical Rules

The lexical elements follow MSC'96. They are identical to the lexical rules for HMSC, see "Summary of Lexical Rules" on page 1740 for further information.

Syntax Rules in Symbols

The descriptions below describes what text is allowed in the different symbols, i.e. it is not a description of the grammar in PR-form.

The grammar below informally describes the lexical units used as terminal symbols. Refer to Z.120 for a complete description.

```
<name> ::= identifier as described above.
<text> ::= approximately the ASCII character set.
```

In the sections below each symbol will be described in the following manner:

1. A *short* description of the symbol and/or its contents and purpose.
2. The symbols textual grammar
3. Possibly any semantic restriction and/or notes. Refer to Z.120 for a complete description of the semantics that apply.

Syntax Common with HMSC

All symbols that can take textual input in the HMSC are also present (and the same) as those found in the MSC. See "Syntax rules in Symbols" on page 1740 for further information.

1. No checks are made on completeness, however.

Syntax Specific for MSC

In addition to the symbols that are common HMSC symbols, the MSC Editor supports the following symbols that contains text.

Comment Symbol Syntax

The text in a comment symbol is an informal explanatory text.

```
<comment text> ::= <text>
```

The text has no (formal) semantic and no restrictions.

Instance, Message and Timer Name syntax

The text identifies the entity

```
<instance name> ::= name
<message name>  ::= name
<timer name>    ::= name
```

Parameter Syntax

Both on messages and instance creations have parameters. In addition the MSC also supports parameters on timers (currently nonstandard). The Z.120 grammar for parameters is defined as follows:

```
<parameter list> ::=
       <parameter name> [ ‘,’ <parameter list> ]
```

This grammar is neither able to allow empty parameter lists, nor to take parameters that are generated from SDL. Furthermore, the grammar does not allow string literals. The MSC Editor (and associated tools) supports the following, more relaxed, grammar:

```
<parameter list> ::=
        [ <relaxed name> { ‘,’ <relaxed name> } * ]
<relaxed name> ::=
        <parameter name> [ ‘(‘  <relaxed name> ‘)’ }
<parameter name> ::=
        <name> |
        <character string> |
        <character string 2>
```

The grammar of `<character string 2>` is the same as the grammar for the Z.120 `<character string>` except that the apostrophe is regarded as the Z.120 `<other character>` and the double quote ‘”’ is regarded as an `<apostrophe>`.

Instance Kind Syntax

The instance kind describes the type of the instance

```
<instance kind>::=
        [ <kind denominator> ] <identifier>
<kind denominator> ::=
        'system' | 'block' | 'process' | 'service'
        | <name>
<identifier> ::= [ <qualifier> ] <name>
<qualifier> ::= '<<' <text> '>>'
```

Decomposition Syntax

This text is present if the instance is decomposed and describes the name of the diagram that explains the decomposed instance in greater detail.

```
<decomposition> ::=
         'decomposed' [ <substructure reference> ]
<substructure referrence> ::=
         'as' <message sequence chart name>
```

Inline Heading Syntax

This is the syntax for the text area in the inline expressions symbols.

```
<inline heading> ::=
        'alt'
        'par'
        'exc'
        'opt'
        'loop' [<loop boundary>]
<loop boundary> ::=
        '<' <inf natural> [ ',' <inf natural> ] '>'
<inf natural>   ::= 'inf' | <natural name>
<natural name>  ::= {0|1|2|3|4|5|6|7|8|9}+
```

Comparing and Merging Diagrams

Overview

Compare and Merge for MSC diagrams

This section describes more details about how the comparison is done in the Compare and Merge functionality for MSC diagrams. The intention is that this should be of interest for a user who wants to understand how the comparison is done and to anticipate what a resulting Compare/Merge will look like.

Compare operation

Two MSC or HMSC diagrams are input to the Compare operation. In general when differences are found related objects are grouped together to form difference groups. In the graphical report it is possible to navigate between these difference groups using the Next and Previous buttons.

For MSC diagrams the algorithms to find difference groups are as follows:

- First the heading is checked
- Then instances are checked
- Then for each instance the events connected to this instance are checked
- Then text symbols are checked
- At last unconnected comment symbols are checked

Comparing Headings

The heading texts will be compared in the two diagrams. If they differ the heading will form a difference group of its own.

Comparing Instances

When instances are compared the algorithm will for each instance try to find a matching instance in the other diagram. To find the match, first of all it uses the name to find the match. If there is no name match, the objectID for the instances will be used to find the match. The objectID is a unique (within each diagram) number that every object has, this is normally not visible for the user but it can be shown using the Info Window. By using objectIDs we will hopefully match instances where the difference is that the name has been changed. When instances have been matched, differences will be reported if any of the three text fields, name, kind and decomposition will differ.

Instances that have not been matched in the two diagrams will then be classified as being added or removed. Each instance will form a difference group of its own.

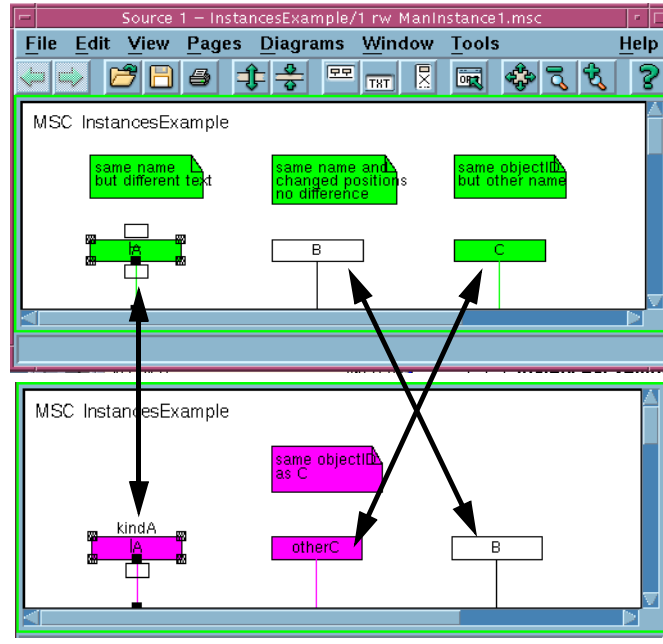


Figure 316 Comparing instances

In Figure 316 on page 1747 two diagrams are shown. When using the Compare command, the instances with the name A will be matched and they will also differ as the instance kind texts are different. Instances with the name B will match and no difference will be detected even though the positions of the instances are not the same. The instance with the name C will be matched and differ against the instance with the name otherC because both have the same objectID. The figure also indicates by the color that all the text symbols differ as either the text has changed or they only appear in one of the diagrams.

Comparing Events

The events will be checked for one instance at a time. All events connected to one instance are ordered from top to bottom. For the corresponding matched instance in the other diagram the event sequences are compared using an algorithm similar to the UNIX diff function for text comparison. The events are group in sequences that have been un-

changed, changed, added or removed. Each such sequence will then form a difference group.

If there is an instance without a match in the other diagrams all the events connected to this instance will be considered as being added and they will all form a difference group.

Special treatment is made for events that are connected to more than one event. For messages the event will be treated together with events connected to the instance where the message starts, except for found messages that are treated on the incoming instance.

Multi instance events like conditions, references and inline expressions are treated in the event sequence for the instance that first encounters the multi event.

After all the event sequences have been checked, difference groups that differ will be sorted from top to bottom according to the Y-position of the objects in the group. Thus, when they are presented in the graphical report they are shown in this order.

Comparing Comments

A comment that is connected to an object will be handled together with that object. This means that if the comment texts differ the whole object will be considered as being different.

An unconnected comment will be matched using the geometrical position of the object or if this check fails the objectID will be used.

Each unconnected comment having differences will form a difference group of its own.

Comparing Text Symbols

The text symbols uses the same algorithms to find differences as the unconnected comments described in the previous section.

Merge operation

The merge operation has two source diagrams as input. The Source2 diagram is merged with Source1 and the resulting diagram is placed in the Result diagram. An optional input is the Ancestor diagram which is used to drive the automerge functionality.

The algorithm for the merge operation is to use the difference groups found when comparing Source1 and Source2. All objects not belonging to any difference group, these are objects being the same in both Source1 and Source2, will be copied from Source1 to Result. For each difference group, the objects in the group are compared with the Ancestor diagram, if no Ancestor is specified the behavior is the same as if the Ancestor diagram was present and it had no objects. If the difference group is a group that has changed and Source1 is the same as the Ancestor, then Source2 is copied to the Result, and if Source2 is the same then Source1 is copied. If none of Source1 or Source2 is the same as Ancestor then the group is classified as being unresolved and the user must decide what the Result should look like.

If a difference group indicates that there are objects in Source1 that have no matching objects in Source2, these objects will be added to Result if they also do not have any matching objects in the Ancestor diagram. However, if there are unchanged matching objects for the whole difference group in the Ancestor they will not be added to Result as in this case they have been removed in Source2. If only parts of them match the group will be classified as unresolved. Similar if a group indicates that objects have been added to Source2 the objects will be added to the Result diagram if the objects also are missing in the Ancestor diagram. If they all have matching objects in Ancestor they will not be added, otherwise the group is set as unresolved.

Figure 317 on page 1750 shows an example of the merge operation.

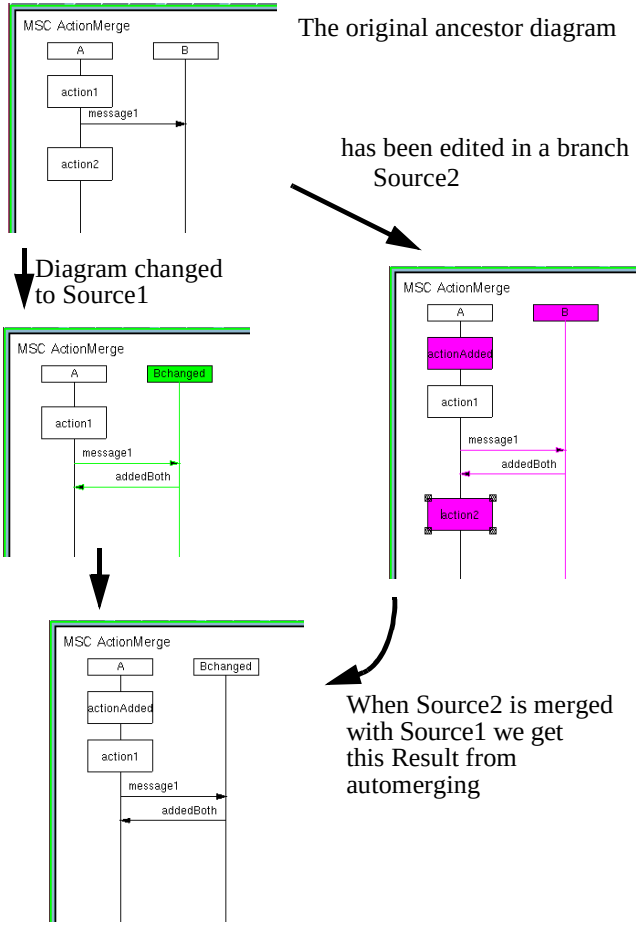


Figure 317 Example using automerge

The automerge gives the following result when the Source2 diagram is merged into the Source1 diagram and the Ancestor diagram has been specified:

- The instance B and Bchanged has been matched and the Result will use the version from Source1 as Source2 and Ancestor are identical.

- The action actionAdded has been added to the Result because there is no match against this in Source2 and it is also missing in Ancestor.
- The action action2 has been removed in Result as it has been removed in Source1 and it was the same in Ancestor and Source2.
- The message addedBoth has been added to Result as it appears both in Source1 and Source2.

Merged diagram with layout changes

During a merge the normal behavior is that, if necessary, a Make Space operation will be performed when a merged object is added to the Result diagram. This will make sure that objects are placed in the correct order and that the diagram will not have overlapping objects.

However, in some circumstances the resulting diagram from a merge operation can have objects placed at positions, which can not be obtained when editing the same diagram by hand. This can happen when the source diagrams do not have much free space in them or when matched objects from the sources are placed at different geometrical positions.

Specifically, please note the following observations:

- As the comparison of the objects is based on the semantic information in the diagram, the geometrical positions of objects can be changed in the merged resulting diagram compared to the positions in the source diagrams. For objects that are the same in both sources and thus being added to the Result diagram, the position from Source1 is used. When objects are inserted from Source2 they will in general not have the same geometry appearance in the Result diagram.
- When objects are merged as the first object on an instance axis they can be placed on the same position and therefore overlap each other. If e.g. both objects are action symbols the visible object can completely hide the object below.
- When messages are merged, the order of events on the given instance will be preserved. This applies to both the in event and the out event. Sometimes this will result in a merged Message object where the message goes “upwards”.

- The geometrical layout of an added difference group might not be same if the Accept/Reject button is pressed several times. However, the semantic information should always be the same.

Merged diagram with unresolved instances

When instances are merged and an unresolved difference is detected for a given instance, all events connected to this instance will be grouped together in one or more difference groups and will be classified as unresolved. If, from the merge dialog, one of the versions of this instance is accepted, the events can also be accepted in later difference groups. If the difference groups will wrap, i.e. by pressing Next many times the difference groups will start from the beginning again, the events connected to this merged instance might not be presented again. They will appear only if there is a real difference for the events. This happens because after a wrap all difference groups are dynamically recalculated and difference groups are first of all found from the differences between Source1 and Source2, and secondly they are formed if any of the instances does not appear in the merged diagram. After the instance has been manually merged the unchanged events connected will then not be part a any difference group.

Performance

The time it takes to perform a Compare/Merge operation depends on how many instances the diagram contains and also on the height of the diagram. There is also a dependence on the type of objects that are used in the diagram, use of multi axis objects like inline expressions, conditions and reference symbols will increase the time used. A 5 pages long diagram with 10 instances and 10 multi-axis objects can take about a minute to perform the Compare/Merge operation.

Unsymmetrical differences can give unexpected difference groups

The use of the diff algorithm when comparing the event sequence on two instance axis can in certain situations give rise to some unexpected results. The diff algorithm looks for the longest sequences of identical events when finding differences. Sometimes this results in two different results for the same sources. Assume Source1 has the sequence ABCD and Source2 has the sequence ACBD. A diff on Source1 and Source2 gives as a result that in Source2 C is unchanged but B has been removed and then B has been added. If instead Source2 is compared with

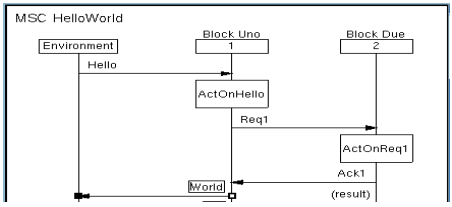
Source1 we will get that B is unchanged and that C has been removed and added. Thus the comparison is not symmetrical.

To compensate for this unsymmetry the resulting difference groups after the diff algorithm are checked for patterns like the one described above. For each unchanged difference group (Ug) the following check is done. If there is a changed or added difference group (CAg) preceding the unchanged one, Ug, and all objects belonging to CAg also can be found in a changed or removed difference group following Ug, the Ug group will also be set as being a changed difference group. Similar if there is a changed or removed difference group (CRg) preceding Ug and all objects belonging to CRg also can be found in a changed or added group following Ug, the Ug group is set as being changed.

Using the example above, we will then get that both B and C will appear in some difference group and thus be considered to have been altered. For the user this can be unexpected, as for example if we only change the diagram by moving object B by dragging it below C we would expect that only B will appear in a difference. However as the compare algorithm rely on semantical information rather than geometrical the tool cannot decide whether it is B or C that has been changed. With the extra check done, we set both as changed and we then will get more difference groups than expected.

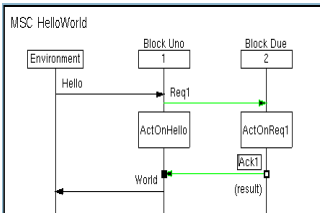
With the extra check made there can still exist some unsymmetry left when comparing. There will be difference groups for the same objects but the difference groups can be of different kinds. This can result, in an automerge situation, that the merged result will not be the same if Source2 is merged into Source1 or if Source1 is merged into Source2. Also the kind of conflicts appearing might be different.

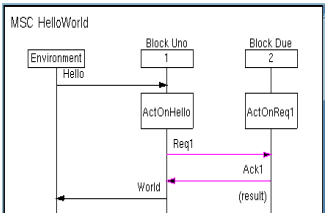
In Figure 318 on page 1754 an example illustrates how this can affect the resulting diagram from an automerge.



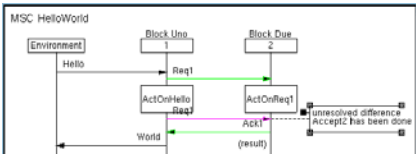
The original ancestor diagram

Source1 and Source2

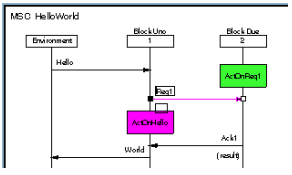




This is what we get when merging Source2 into Source1



This is what we get when merging Source1 into Source2



Here there will be a conflict for the Req1 message. The diagram shows the result after Accept2 but the diagram will have the same appearance if Accept1 has been used instead.

Figure 318 Result from automerge

From the changes in Source1 and Source2 it is uncertain what the result of a merge will look like. Is the changed that the message Req1 will be either after or before the action symbols? or is the change that the action symbols are moved relative to the message? Both expectations could be justified and in the automerge, both when merging Source2 into Source1 or Source1 into Source2, we will get a merge conflict (however

not the same conflict), which signals the user that he must check this more carefully.

The conclusion is that you should be careful when using automatic merge, sometimes your expectations will not come true.

Compare and Merge

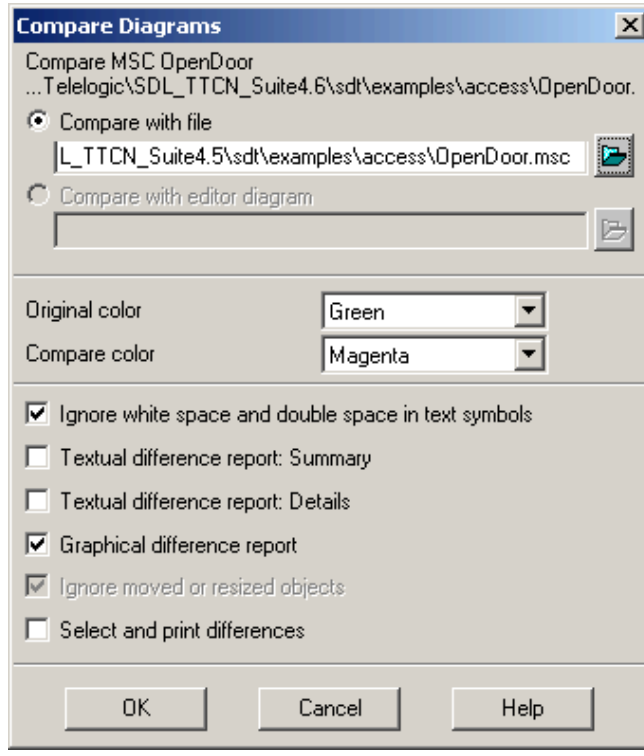


Figure 319: The Compare Diagrams dialog in the MSC Editor

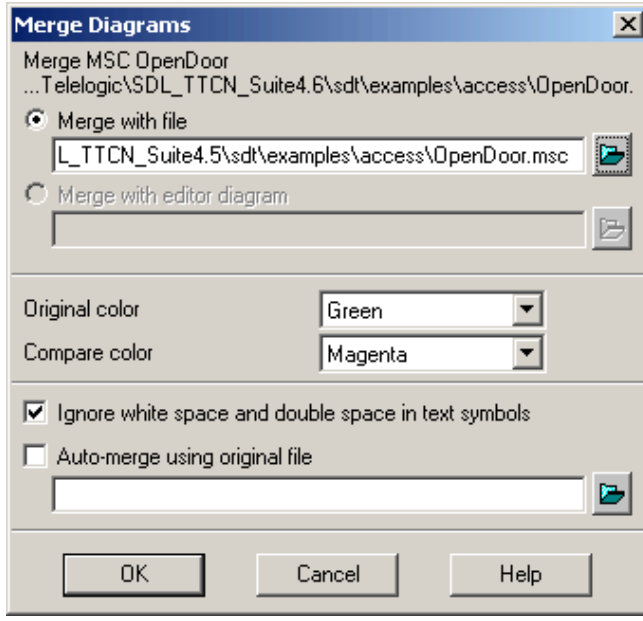


Figure 320 The Merge Diagrams dialog in the MSC Editor

When you select *Compare Diagrams* or *Merge Diagrams* from the *Tools* menu in the MSC Editor or in the HMSC Editor, a dialog will be opened where you can select the diagram to compare with. (It is also possible to compare more than one diagram pair in one operation, but to do this you should use the corresponding menu choices in the Organizer. See "Compare > MSC Diagrams" on page 151 in chapter 2, *The Organizer*, "Compare > HMSC Diagrams" on page 151 in chapter 2, *The Organizer*, "Merge > MSC Diagrams" on page 151 in chapter 2, *The Organizer* and "Merge > HMSC Diagrams" on page 152 in chapter 2, *The Organizer*.

Specifying the Diagram to Compare With

There are two ways for specifying the diagram that should be compared with the currently displayed diagram:

- You may specify the file in the *Compare with file* field.

- You may specify another diagram already opened in the editor in the *Compare with editor diagram* field. When you click the folder button, a dialog will be opened with a list of all diagrams opened in the editor (except for the currently displayed).

  The option *Compare with editor diagram* has the two restrictions:

  – It can only be used if more than one diagram is currently opened in the editor.

  – The folder button can only be used if more than two diagrams are loaded into the editor.

Selecting the Report Type

In the Compare Diagrams dialog, there are six alternatives for reporting differences between diagrams: *Ignore white space and double space in text symbols*, *Textual Difference Report: Summary*, *Textual Difference Report: Details*, *Graphical Difference Report (with Merge Facility)* and *Select and Print Differences*.

The Merge Diagrams dialog does not allow you to get a textual difference report. Instead, you always get a graphical difference report and the only option available is *Auto-merge using original file*.

Ignore white space and double space in text symbols

When this option is on, differences caused by different usage of white space are ignored when comparing texts. One space character is considered equal to two space characters.

Textual Difference Report: Summary

This option is only available when comparing using the *Compare Diagrams* menu choice.

A summary of how many differences that has been found is presented in the Organizer log. The summary may look like this:

```
Comparing /home/lat/x.spr
     with /home/lat/y.spr
27 differences (16 symbols, 11 lines) :
  7 changed objects (7 symbols, 0 lines)
  13 removed objects (6 symbols, 7 lines)
  7 added objects (3 symbols, 4 lines)
```

When you read this report, notice the following:

- 27 is the total number of found differences in this diagram pair. This is the total of all changed, removed and added objects.
- An object is considered to be changed if you have moved or resized it, or if you have changed a text associated with the object.

If at least one of the two compared diagrams has more than one page, then a summary is also presented for each page pair. It may look like this:

```
Comparing /home/lat/version1/registeredcard.spt
     with /home/lat/version2/registeredcard.spt
5 differences (1 symbol, 4 lines) :
  1 changed object (1 symbol, 0 lines)
  2 removed objects (0 symbols, 2 lines)
  2 added objects (0 symbols, 2 lines)

Page details:

Page ValidateCard
5 differences (1 symbol, 4 lines) :
  1 changed object (1 symbol, 0 lines)
  2 removed objects (0 symbols, 2 lines)
  2 added objects (0 symbols, 2 lines)

Page RegisterMode
0 differences
```

Textual Difference Report: Details

This option is only available when comparing using the *Compare Diagrams* menu choice.

This option will print information about each found difference in the Organizer log. This makes it possible to navigate from the Organizer log to every found difference by using the *Show Error* operation in the Organizer log. See "Organizer Log Window" on page 183 in chapter 2, *The Organizer*.

The detailed textual difference report may look like this:

```
Comparing /home/lat/d1/db.sbk
     with /home/lat/d2/db.sbk

Comparing page 1

#SDTREF(SDL,/home/lat/d1/db.sbk(1),119(50,35))
ERROR This symbol has been changed...
#SDTREF(SDL,/home/lat/d2/db.sbk(1),119(50,40))
ERROR ...and now looks like this.
```

```
#SDTREF(SDL,/home/lat/d1/db.sbk(1),176(95,60))
ERROR This symbol has been removed.

#SDTREF(SDL,/home/lat/d2/db.sbk(1),182(60,60))
ERROR This line has been added.
```

Graphical Difference Report (with Merge Facility)

The merge facility is only available when merging with the *Merge Diagrams* menu choice, but you can get a graphical difference report both when comparing with the *Compare Diagrams* menu choice and when merging with the *Merge Diagrams* menu choice.

Two windows, named *Source 1* and *Source 2*, displaying the two diagram versions and their differences, and one dialog will be opened. For merge, there will be a third window, named *Result*, showing the merge result.

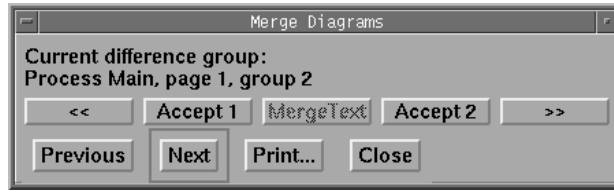


Figure 321: The Merge Diagrams dialog

The Compare Diagrams dialog is similar to the Merge Diagrams dialog. The difference is that the Compare Diagrams dialog is missing the first row of buttons. (“<<“, Accept1, Accept2 and “>>”.)

In the dialog that will be opened, you can:

– Navigate between difference groups with the *Next* and *Previous* buttons.

– (*Merge Diagrams* only.) Navigate between unresolved (not-yet merged) difference groups with the “<<“ and “>>” buttons.

– (*Merge Diagrams* only.) Do a merge, i.e. update the merge result diagram with information from the current difference group

and one of the compared diagram versions, with the *Accept1/Reject1* and *Accept2/Reject2* buttons.

– Compare or merge a text in a text symbol in the current difference group.

– Print the current difference found in the current page pair.

When the second compare dialog is closed, the editor windows are restored to how they appeared before the compare operation.

Comparing text in text symbols

When two text symbols differ the *CompareText* button will be available in the Compare Diagrams dialog and it will be possible to enter the Compare Text Symbols dialog.

The Compare Text Symbols dialog consists of two lists of text that are placed side by side with a "gutter" that will contain extra information about the differences. This gutter-information consist of:

– A "-" indicates that there is no corresponding line in the text compared to the other text.

– A "|" character indicates that this line differs in the two texts.

The current difference is indicated with a highlight. Buttons make it possible to navigate to *Previous* and *Next* difference item.

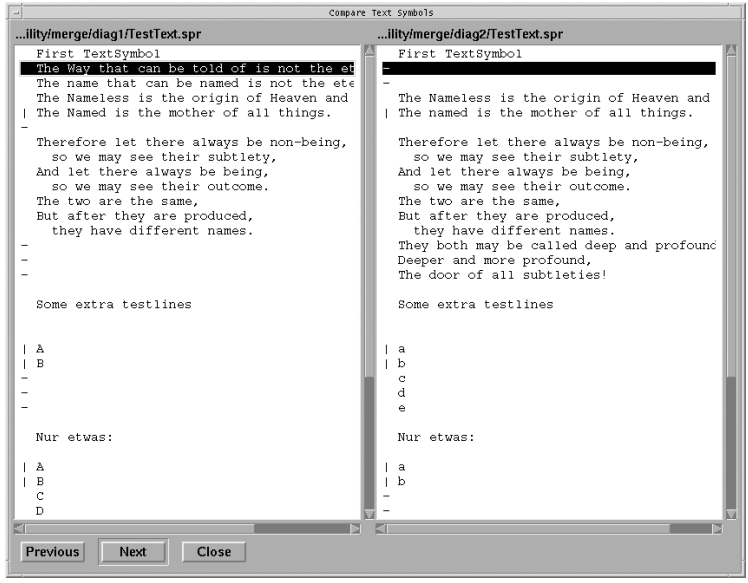


Figure 322: The Compare Text Symbols dialog

Merging text in text symbols

When manual merges are conducted the option to merge text symbols will always be available in the Merge Diagrams dialog. When merging a *MergeText* button is available as in the compare function. When a text symbol is selected there will be an option to select either symbol or merge the text in the symbols into a new symbol.

If the text merge is selected a Merge Text Symbols dialog is presented with three text lists, the two source texts and the result text. It is possible to navigate between the unresolved differences or between all differences.

Figure 323: The Merge Text Symbols dialog

This works in the same way as in the normal merge. The buttons "<<" and ">>" will jump between the unresolved differences and the *Previous* and *Next* buttons will jump between all differences.

The use of *Accept 1*/*Reject 1*, *Accept 2*/*Reject 2* buttons will select neither, either or both the texts for merge. The selections from the source files are presented in the result-list the gutter will show what version is selected. Additional symbols in the result list are:

- A "|" character indicates that this line differs in the two texts.
- A "=" character indicates that the selected text are equal in both versions.
- A ">" character indicates that the right text has been selected for merge.
- A "<" character indicates that the left text has been selected for merge.

It is also possible to edit the result line-by-line by selecting a line and edit it below the list. Clicking the *Change* button will replace the changed line in the result list.

Ignore moved or resized objects

If this option is on, the compare operation tries to ignore differences only caused by moved or resized symbols. You should only use this option if the two compared diagrams originates from the same diagram. (For instance, you have used Save as on the original diagram to get the second copy.) This option will not work well if you create two diagrams from scratch and compare them, because the operation uses object IDs to match moved or resized objects.

Select and Print Differences

If this option is on and there are differences, then a print dialog will pop up when the first dialog is closed. Only diagram pages containing differences will be printed, and the differences will be highlighted with selection markers. If the same page containing differences can be found in both compared diagrams, then the two pages will be printed after each other, beginning with the page from the diagram that was visible in the editor when the compare operation was invoked.

Auto-merge using original file

This option is only available in the Merge Diagrams dialog. It allows you to specify the original diagram file that both the diagram versions that are compared originates from. If you specify the original diagram file, then the merge operation can auto-merge all non-conflicting changes for you. Note that if you give the wrong original diagram, then the auto-merge operation may merge the two versions in the wrong way. If you do not specify an original diagram, then the merge operation leaves all merging of differences to you.

Differences That Will Be Reported

When you use *Compare Diagrams* or *Merge Diagrams*, all visual object differences will be found:

- An object that has been moved, resized, removed, added or painted in a different color.
- A text, associated with an object, that has been changed

An object is considered to be moved if it has been moved relative to the upper left corner of the diagram frame. This means that if you move the frame, it will have no effect.

Diagrams are compared one page pair at a time. A page pair is one page from the original diagram and one page from the other diagram where both pages have the same name. If there is no page with the same name in the other diagram, then the complete page is considered to be a difference.

Difference Groups

Differing objects from the same page pair are grouped into *difference groups*. Two differing objects are placed in the same group if:

- The objects are from the same diagram and they are connected to each other. For instance, a flow line connected to a task symbol.
- The objects are from different diagrams and they would overlap if they were in the same diagram, that is, it would be impossible to place the two objects in the same diagram without changing the position within the diagram for one of them.
- The objects are from different diagrams and the difference operation can conclude that it is the same object that has been moved.

Color and the Current Difference Group

In the graphical difference report, all differing objects are given temporary background colors as specified in the initial dialog. As default, the color is green in the original diagram and magenta in the other diagram. Temporary colors are not saved in diagram files. How temporary colors are removed is described in "Temporary Colors > Remove" on page 1675.

The dialog that will be opened during the graphical difference report, allows you to navigate between difference groups using previous and next buttons. All objects in the current difference group are selected.

Normally, version 1 of the compared diagrams (the diagram that was in the editor when the compare operation was invoked) is shown in a window in the top left corner of the screen. Version 2 (the diagram that was specified in the first compare dialog) is shown in the top right corner off the screen. For complete page differences (a difference where a page can not be found in the other diagram), the window that misses the page is hidden.

Note that it is possible to manually edit the compared diagrams while the compare operation is ongoing. This can be seen as a light-weight alternative to the full merge operation (described next).

Merging

During a merge operation, three windows are used in addition to the second merge dialog:

- The top left corner of the screen is used to show version 1 of the compared diagrams, in the editor window named *Source 1*. This is the diagram that was visible in the editor window when the merge operation was invoked.
- The bottom left corner of the screen is used to show version 2 of the compared diagrams, in the editor window named *Source 2*. This is the diagram that was specified in the first merge dialog as the diagram to compare with.
- The right part of the screen is used to show the merge result diagram, in the editor window named *Result*. This is a new diagram that has been created from the two compared diagram versions.

The second Merge dialog does not only have previous and next buttons to navigate between differences. It does also have *Accept/Reject* buttons and “<<” (fast backward) / “>>” (fast forward) buttons.

- Pressing *Accept 1* (*2*) will add symbols to the merge result diagram. It is the symbols from version 1 (2) of the current difference group that will be added.
- Pressing *Reject 1* (*2*) will remove symbols from the merge result diagram. It is the symbols that can be found in version 1 (2) of the current difference group that will be removed.
- Pressing “<<” (fast backward) will navigate to the previous unresolved difference group. An unresolved difference group is a difference group where both version 1 and version 2 are not accepted.
- Pressing “>>” (fast forward) will navigate to the next unresolved difference group.

Note that it is possible to manually edit diagrams while a merge operation is ongoing. This can be useful if you want the merge result diagram to look different from both version 1 and version 2 of the compared diagrams.

When the second merge dialog is closed, the editor will ask you to save the merge result by displaying a save dialog above the window with the

merge result diagram buffer. When the save dialog is closed, the editor windows are restored to how they appeared before the merge operation was invoked.

Literature References

Object Model Literature References

There is a growing number of publications devoted to object modeling techniques. The following two sources have been selected as defining the Object Model supported by the OM Editor:

[6] OMG Unified Modeling Language, version 1.3, June 1999
(document ad/99-06-08) Final Draft
Object Management Group
ftp://ftp.omg.org/pub/docs/ad/

[7] Object Oriented Modeling and Design
Rumbaugh, Blaha, Premerlani, Eddy, Lorensen
Prentice-Hall (1991)
ISBN 0-13-630054-5

State Chart Literature References

There is a growing number of publications devoted to state chart techniques. The following source has been selected as defining the State Chart supported by the SC Editor:

[8] Unified Modeling Language, version 1.1, Sept. 1997
UML Semantics (document ad/97-08-04)
UML Notation Guide (document ad/97-08-05)
Object Management Group
ftp://ftp.omg.org/pub/docs/ad/

[9] Unified Modeling Language, version 1.0, January 1997
UML Semantics (document ad/97-01-03)
UML Notation Guide (document ad/97-01-09)
Object Management Group
ftp://ftp.omg.org/pub/docs/ad/

MSC‘96 Literature References

[10] Z.120 (1996).
Message Sequence Chart (MSC).
ITU-T, Geneva, April 1996

[11] Message Sequence Chart
Syntax and Sematics
M.A. Reniers, Eindhoven University of Technology, 1999
IPA Dissertation Series 1999 - 7

[12] Object-Oriented Software Engineering
– A Use Case Driven Approach.
I. Jacobson.
Addison-Wesley, 1992

[13] Tutorial on Message Sequence Charts (MSC’96)
Ekkart Rudolph, Jens Grabowski, and Peter Graubman

UML Literature References

[14] The Unified Modeling Language User Guide
Grady Booch, Ivar Jacobson, James Rumbaugh.
Addison-Wesley, 1998

[15] The Unified Modeling Language Reference Manual
James Rumbaugh, Ivar Jacobson, Grady Booch.
Addison-Wesley, 1998

[16] The Unified Software Development Process
Ivar Jacobson, James Rumbaugh, Grady Booch.
Addison-Wesley, 1999

Chapter

40 *The Deployment Editor*

The Deployment Editor is a tool for graphical partitioning of SDL systems, which allows you to model how SDL systems execute in their target environment. See also chapter 64, *Integration with Operating Systems*.

Diagrams in the editor, deployment diagrams, use notation from UML. Information from a deployment diagram can be used when targeting one or several SDL systems. See chapter 59, *The Targeting Expert*.

This chapter contains a description of the deployment diagram as well as guidelines for modeling different deployment scenarios. The connection with the Targeting Expert is also described.

Introduction

The Deployment Editor allows you to graphically deploy one or several SDL systems. SDL systems can be partitioned into components, i.e. executable files, which in turn are partitioned into threads of execution. Finally, SDL entities, i.e. objects, are mapped to the threads, showing the execution architecture at compile-time.

The information in a deployment diagram can be used by the Targeting Expert through the Partitioning Diagram Model. In the Targeting Expert, you can set build parameters on each component and build an application from this model.

Applications

You can use the Deployment Editor in two ways:

- As an editor for graphical partitioning of SDL systems.
- As a pure modeling tool.

The Deployment Editor gives you freedom in modeling partitioning and communication. There are few restrictions in drawing rules and no syntax checking is performed. There are, however, restrictions how to use deployment diagrams for generating partitioning data for the Targeting Expert. This chapter describes the requirements for using the Deployment Editor for partitioning. The diagram symbols are described as they are interpreted when used for partitioning.

Starting the Deployment Editor

You start the Deployment Editor the same way as the other editors in the SDL Suite. There are four alternatives:

- Select *Tools -> Editors -> Deployment Editor* in the Organizer to start with an empty diagram.
- Select *Edit -> Add New...* in the Organizer and *UML -> Deployment* in the *Add New* dialog to add a new diagram.
- Select *Edit -> Add Existing...* in the Organizer to add an existing diagram.
- Right-click on an existing deployment diagram symbol in the Organizer and select *Edit -> Edit* in the pop-up menu.

The Graphical User Interface

The graphical user interface in the Deployment Editor is similar to that used in the other SDL Suite editors. For a description of the user interface in general, see chapter 1, *User Interface and Basic Operations, in the User´s Manual*.

Connection to the Targeting Expert

The Deployment Editor generates data that can be used by the Targeting Expert. Deployment diagrams are translated to the data format Partitioning Diagram Model (PDM), which contains build-related information. See "Generating Partitioning Diagram Data for the Targeting Expert" on page 1787.

To start the Targeting Expert with PDM data for a deployment diagram:

1. Click on the deployment diagram (.sdp file) in the Organizer
2. Select *Targeting Expert* from the *Generate* menu.

See chapter 59, *The Targeting Expert, in the User´s Manual* for information on how to build using the Targeting Expert.

Using Information from SDL System(s)

The Deployment Editor has no automatic access to information about SDL systems in the Organizer. You must enter information about SDL objects manually in a deployment diagram.

For SDL objects, the following information is mandatory to enter:

- *Name*

  It is recommended that the name of the object in the Organizer is used.
- *Qualifier*

  This is the *path* to the SDL object in the Organizer. The qualifier should include the name of the object in the Organizer.
- *Stereotype*

  This shows the type of SDL object and should have one of the following values: `system`, `block` or `process`.

See "Object" on page 1777 for more information about SDL objects.

Note:

It is possible to deploy objects from many SDL systems in one deployment diagram. The only requirement is that the SDL systems must be contained within the same `.sdt` file. This `.sdt` file must be loaded in the Organizer when working with the deployment diagram.

The Deployment Diagram

In this section, the deployment diagram is described. The view and the symbols as well as drawing rules for the different integration models are explained.

Note:

Following the naming convention in the SDL Suite, diagrams edited in the Deployment Editor are named deployment diagrams. This notion must **not** be mistaken for the UML deployment diagram. In this chapter, “deployment diagram” implicitly means the SDL Suite diagram.

Note:

In the Deployment Editor, it is only possible to use one page per diagram. This differs from the OM Editor, where an arbitrary number of pages can be used. See “OM Editor Specific Information” on page 1691 in chapter 39, *Using Diagram Editors, in the User´s Manual*.

The View

The deployment diagram uses a view that is similar to the UML component view. The most important similarities are:

- Both views are type-based, i.e. no instance-specific information can be modeled. This implicates that the thread and object symbols in the Deployment Editor diagram can be regarded as stereotyped UML class symbols.
- Both views are static, i.e. partitioning at compile-time is modeled.

There are some important differences between the Deployment Editor view and the component view:

- The Deployment Editor view uses the node symbol, which is not used in the UML component view. In UML, the node symbol is only used in the deployment view.
- The Deployment Editor view does not support the interface notion, which is used for modeling communication between components in the UML component view.

The Deployment Editor diagram shows a static view of the partitioning, i.e. you can deploy SDL systems by how they look at compile-time.

The Symbols

The deployment diagram contains four symbols that can be used for modeling partitioning: Node, Component, Thread and Object. Each symbol is described in the following subsections. See "About Symbols and Lines" on page 1635 in chapter 39, *Using Diagram Editors, in the User´s Manual*.

> **Note:**
>
> The deployment diagram terminology must not be mixed up with the syntax used in buildscripts, i.e. scripts containing SDL Analyzer commands. Some terms, such as *thread*, are similar, while other, such as *component*, represent completely different things.

All symbols, except the thread symbol, contain information that can be edited in the *Symbol Details* dialog box. Properties and stereotype information are graphically visible on the symbols.

Node

A node symbolizes a computational resource, i.e. something with processing power and possibly some memory and other peripherals. A node should be seen as a platform on which software execute. Only components can be attached to nodes.

A typical stereotype for a node is a description of the processor and a common property is the operating system that is being used. Figure 324 shows a node called **My_Computer** with the stereotype **PC** and the property **Windows NT 4.0**.

Figure 324 A node with name, stereotype and properties

For a node, you can set the stereotype to *external*, indicating that the node does not execute any code from your SDL system(s). This makes it possible to model external non-SDL parts that communicate with your SDL system(s). Setting the stereotype to `external` implicates that the node is ignored when building a Partitioning Diagram Model from the deployment diagram.

Note:

Stereotype and property information for a node is ignored when generating partitioning diagram data. The only exception is when stereotype is set to `external`.

Component

A component symbolizes an executable file, i.e. a sort of container for software entities. Components contain objects, i.e. SDL systems, blocks or processes, that are organized in threads of execution. You can attach either objects or threads to a component.

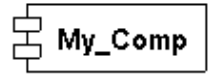


Figure 325 A component symbol named "My_Comp"

As for nodes, you can set `external` as stereotype on a component, which means that the component does not contain any SDL object from the system(s) that are deployed.

Note:

Stereotype and property information for a component is ignored when generating partitioning diagram data. The only exception is when stereotype is set to `external`.

For a component, you can select an integration model in the *Symbol Details* dialog box. The integration model reflects how the code that is generated is integrated with your target operating system. Three integration models are available:

- *Light Integration.* This is the default selection.
- *Tight Integration.*

- *Threaded Integration.*

See "Integration Models" on page 1780 for more information.

Thread

The thread symbol symbolizes a static thread of execution at compile-time. Threads can only be attached to components. Only objects, i.e. SDL entities, can be attached to a thread.

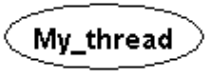


Figure 326 A thread symbol named "My_thread"

The thread symbol can only be used for modeling threaded integrations, i.e. for a Deployment Diagram with threads. Partitioning Diagram Data can only be generated if the component has "Threaded" as integration model.

You can make four settings for a thread through the "Symbol Details" dialog box. These are:

- *Thread Priority*
- *Stack Size*
- *Queue Size*
- *Max Signal Size*

The settings are available to the Targeting Expert through the Partitioning Diagram Model. The settings are common among real-time operating systems.

The values that are entered are platform-specific and are used when compiling the system for your target environment. You are free to enter any value, e.g. a number or a macro, that makes sense in your target environment.

You can choose to either explicitly set a value, or leave it empty. Any setting that is not filled in is considered undefined. In that case, a default value is used.

Note:

Your target operating system may have thread settings that do not exactly match the available thread settings in the Deployment Editor. If you know that a setting is not available in your Operating System, leave the text box empty.

Object

The object symbol is a general notion of an SDL system, block or process. The stereotype field is used to state the type of SDL symbol. This information is mandatory and has three possible values:

- `system`
- `block`
- `process`

The stereotype value is shown within the object symbol. Figure 327 shows an object with `process` as stereotype.

Figure 327 An object symbol with name and stereotype

Note:

It is recommended that the object name in the deployment diagram correspond to the name of the SDL object which is deployed. However, it is not required that the names are identical. For instance, if you add two SDL objects with identical SDL names to the same namespace, you might want to separate them by changing the object name of one the SDL objects. You can freely change the object name, as the object is uniquely identified by its qualifier.

For an object, it is mandatory to enter information in the Qualifier field in the Symbol Details dialog box. The qualifier consists of the *path* for an SDL object in the Organizer tree structure. '/' is used as a separator. The path must include the object name itself.

Some examples of object qualifiers for the AccessControl system are shown in the table below. You can verify the qualifiers by opening the AccessControl system in the Organizer (`< SDL Suite installation directory>/examples/AccessControl/Accesscontrol.sdt`) and comparing them with the entries in the table.

Object Name	Qualifier
CodeReader	`AccessControl/Local/CodeReader`
Controller	`AccessControl/Local/Controller`
Central	`AccessControl/Central`
Display	`AccessControl/Local/Display`
Doors	`AccessControl/Local/Doors`

An example of an object with all mandatory information entered is given in Example 316.

Example 316: Object Name, Qualifier and Stereotype

In the AccessControl system, the process *Central* in block *CentralUnit* has `AccessControl/CentralUnit/Central` as qualifier. For readability, the object is given the name `Central`. The stereotype is set to `process`.

Associations and Aggregations

The Deployment Editor diagram uses associations for modeling relations between nodes. Composite aggregation is used for the following relations:

- Node - Component
- Component - Thread
- Component - Object
- Thread - Object

There is a *Line Details* dialog box, which is common for all sorts of relations. For an association, you can model protocol, encoding and direction data. For a composite aggregation, only multiplicity data can be edited. For a more detailed description of the dialog box, see "DP Editor Specific Information" on page 1711 in chapter 39, *Using Diagram Editors, in the User´s Manual*.

> **Note:**
>
> Associations between nodes are ignored when partitioning diagram data is generated.

Integration Models

The Deployment Editor supports three different integration models. The choice of integration model is set at component level.

Light Integration

In the light integration model, one or many SDL objects execute within one single task. The integration model is called *Light* in the *Symbol Details* dialog box.

The thread symbol is not used when modeling a light integration. All objects that should execute in a component with a light integration should be attached directly to the component. See Example 317 for an example of a light integration.

Example 317: A Light Integration ----------

This example (see Figure 328) shows a light integration containing two objects from the *AccessControl* system. All process instances run in one single task in one executable file. The objects are connected directly to the component, which has *Light* as integration model. The `Control` block has `AccessControl/LocalStation/Control` as qualifier and the `DoorControl` block has `AccessControl/LocalStation/DoorControl` as qualifier. Both objects have `block` as stereotype.

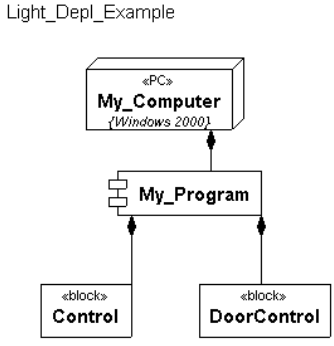


Figure 328 A light integration

Tight Integration

In a tight integration, each SDL process instance is put in a task of its own. The number of tasks varies at run-time, as process instances are created and terminated dynamically.

As described in "The View" on page 1773, the deployment diagram shows a static view of SDL systems at compile-time. The thread symbol symbolizes one static thread of execution, which makes it inappropriate for modeling tight integrations. Therefore, the thread symbol is not used for modeling tight integrations. Always attach objects directly to a component when you model a tight integration.

> **Note:**
>
> The Deployment Editor does not prevent you from attaching threads to a component with *Light* or *Tight* as integration model. The check is performed when partitioning diagram data is generated for the Targeting Expert. If you have an invalid diagram, you will get error messages in the Organizer Log.

An example of a typical tight integration is shown in Example 318.

Example 318: A Tight Integration ––––––––––––––––––––––––––––

This example (see Figure 329) shows a tight integration where two processes from the AccessControl system are executing in one component. The component is deployed on a node.

Each process instance of the two SDL processes will execute in an OS task of its own on the target operating system.

The component has *Tight* as integration model (not visible in the figure). DoorController has `AccessControl/LocalStation/DoorControl/DoorController` as qualifier and `process` as stereotype. Controller has `AccessControl/LocalStation/Control/Controller` as qualifier and `block` as stereotype.

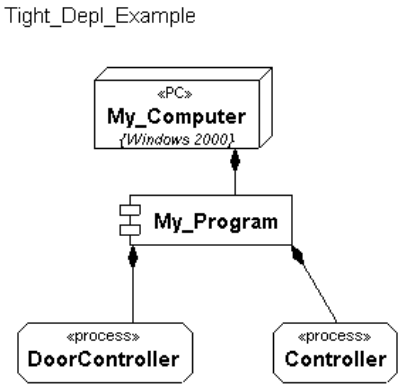


Figure 329 A tight integration

Threaded Integration

Using the threaded integration model, you can map an arbitrary number of SDL objects to an arbitrary number of threads. Each thread symbolizes a light-weight process in the executable file that is generated.

All objects that are connected directly to a component, having a threaded integration, are considered executing together in one implicit thread. Only default values for thread priority, stack size, etc. are available for an implicit thread.

The most common deployment case is that the smallest, indivisible entity is the process instance set. This is the smallest entity that can be graphically represented in a Deployment Diagram. However, it is possible to make each single instance of an object execute in a separate thread in the threaded integration model. This is modeled using multiplicity on the aggregation between a component and the thread that contains the SDL object. Set the multiplicity of the aggregation between the component and the thread to '*' to model this scenario. For a thread with '*' as multiplicity, the values of thread priority, stack size and queue size, etc. will apply to all threads that are created in the target environment.

A complete example of a threaded integration that illustrates the concepts is shown in Example 319.

Example 319: A Threaded Integration ––––––––––––––––––––––––

Figure 330 shows a component with many threads attached. The thread `T` has two objects (`CentralUnit` and `PanelControl`) attached to it. All process instances in these two blocks will execute within one thread. The component should execute on Windows 2000. Thus, Windows-specific thread settings are given. The following values are set: Thread Priority: `THREAD_PRIORITY_ABOVE_NORMAL`, Stack Size: `2048`. The other fields are left empty, as default values should be used.

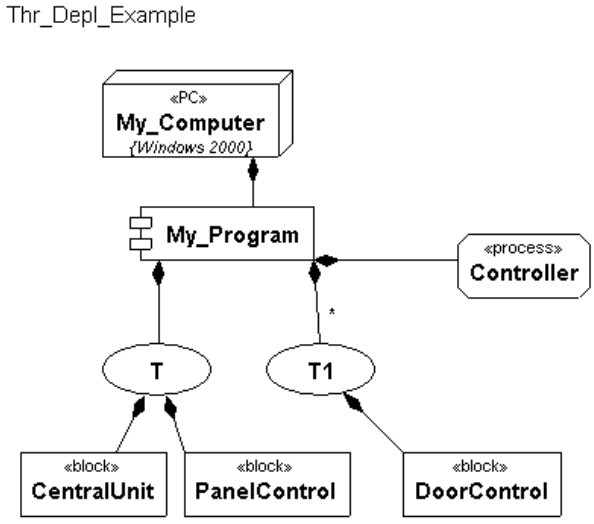


Figure 330 A threaded integration

The `Controller` object is attached directly to the component and will thus execute in an implicit thread. If there were more objects attached directly to the component, they would execute in the same thread as `Controller`. As the thread is implicit, default values are used in the thread settings.

The thread `T1` has '*' as multiplicity in the aggregation between itself and the component. This indicates that all process instances in objects attached to `T1` will execute in a thread of its own. The thread settings for

`T1` are: Priority: `THREAD_PRIORITY_BELOW_NORMAL`, Stack Size: `1024`. These settings will apply to all threads that are created.

All objects have stereotype and qualifier values set as shown in the previous examples.

Deployment Workflow

Drawing a Deployment Diagram

One or many SDL systems can be deployed in the same deployment diagram. However, it is necessary that all the deployed SDL systems reside in the same `.sdt` file.

Perform the following steps to draw a complete diagram:

1. Start the Deployment Editor with a new diagram. See "Starting the Deployment Editor" on page 1770 for instructions.
2. Name the diagram. As the diagram name is used for naming the target directory root, it is important to select a descriptive name.
3. Identify the nodes on which your system(s) will execute. Draw the nodes in the deployment diagram.
4. Name each node and edit stereotype and properties values in the *Symbol Details* dialog box, if desired.
5. For each node, identify the executables. Each executable is symbolized by a component.
6. Draw the components in the deployment diagram and draw aggregations from the node symbols to the components that should execute on them.
7. Set name and edit the integration model for each component.
8. Draw the threads that you wish to use and enter thread settings for each thread.
9. Draw the objects that you wish to deploy. Collect name, qualifier and stereotype information from the Organizer view of your SDL systems and enter this information in each object.
10. Connect each object to a component or thread, depending on the integration model chosen.

> **Tip:**
>
> Let the Organizer window be visible when you edit the diagrams for easy access to SDL object information.

Example 320 shows a deployment scenario where components with different integration models are combined.

Example 320: AccessControl Deployment ----------

This example shows a Deployment Diagram for the AccessControl system. The complete system is deployed on two nodes, as shown in Figure 331. On each node there is a component. The component `Prog1` has Threaded as integration model and uses threads. See Example 319 for a more detailed description of threaded concepts.

The node named `Ethernet_Hub` is contained in the diagram to show how the nodes are physically connected to its external environment in a network. As this node has `external` as stereotype, it is ignored when partitioning diagram data is generated. The associations between the nodes are also ignored.

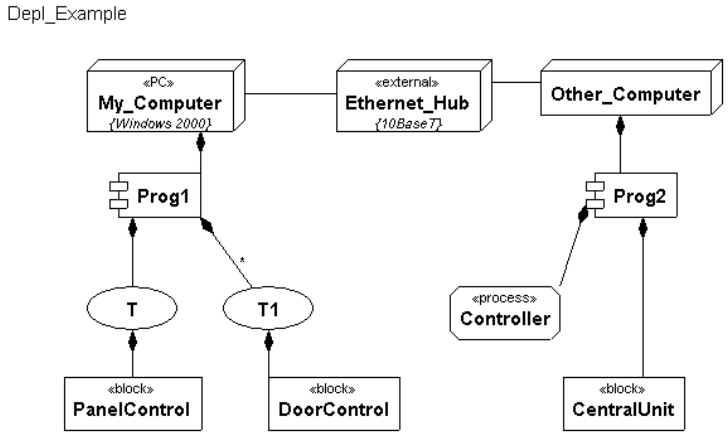


Figure 331 Deployment of the AccessControl system

The component `Prog2` has *Tight* as integration model. The SDL objects are attached directly to the component. Note that the integration model is not visible in the diagram. It is set in the “Symbol Details” dialog box.

Tip:

Draw several Deployment Diagrams to experiment with different deployment situations for your SDL systems. This can be useful when you experiment with different thread configurations.

Generating Partitioning Diagram Data for the Targeting Expert

The Targeting Expert uses the Partitioning Diagram Model (PDM) to structure an application that should be built. It uses the node and component concepts from the Deployment Diagram.

PDM files can be generated from Deployment Diagrams using the Organizer. In order to start the Targeting Expert with partitioning diagram data from a deployment diagram, perform the following tasks:

1. Right-click the deployment diagram symbol (`.sdp` file) in the Organizer.
2. Select *Targeting Expert* from the pop-up menu.

A partitioning diagram model is now generated. If there are errors or information messages, the Organizer Log is opened. If the generation passes, the Targeting Expert is started in interactive mode with the partitioning diagram.

The PDM file is saved in the same directory as the `.sdp` file and has the name as the `.sdp` file except for the extension, which is `.pdm`. As the PDM data is saved on file, it can be used by the Targeting Expert both in interactive mode and batch mode.

Note:

When the Targeting Expert is started in interactive mode with a deployment diagram, the `.sdt` file that contains the referenced SDL objects must be loaded into the Organizer. The Targeting Expert cannot be started from a deployment diagram without an `.sdt` file which has been saved.

PDM in Targeting Expert Interactive Mode

To start the Targeting Expert in interactive mode with data from a deployment diagram, you must right-click the .sdp file symbol in the Or-

ganizer and select Targeting Expert. The Partitioning Diagram Model is not visible from the Organizer.

The Partitioning Diagram Model for the deployment diagram is shown in the upper-left corner in the Targeting Expert window. The application is seen on node and component level. For each component, you make build settings. Depending on the integration model selected in the deployment diagram, specific predefined integration settings are available.

For more information on the Targeting Expert batch mode, see "Interactive Mode" on page 2910 in chapter 59, *The Targeting Expert*.

PDM in Targeting Expert Batch Mode

The Targeting Expert can use PDM files in batch mode by either setting the flag `-pdm` or by giving the command `Open-PDM` as a batch mode command. A PDM file from a Deployment Diagram must be generated using the Organizer. After that, the PDM file can be used in either interactive or batch mode.

For more information on the Targeting Expert batch mode, see "Batch Mode" on page 2958 in chapter 59, *The Targeting Expert*.

Chapter

41 *Editing MSC Diagrams*

This chapter describes how to create and edit MSCs (Message Sequence Charts).

If you only have a MSC Trace license this chapter is still applicable since you are allowed to move symbols in a generated trace. Adding and removing symbols is however blocked.

For a reference to the MSC Editor, see chapter 39, *Using Diagram Editors*.

General

Editing Functions

The MSC Editor is the graphical tool that you use to create, edit, print and store Message Sequence Charts. The MSC Editor provides, among other things, the facility of including MSCs into SDL systems.

The basic editing functionality that the MSC Editor provides can be summarized as:

- Creating and managing MSCs
- Appending, moving, resizing symbols
- Interconnecting symbols with lines
- Editing text
- Printing MSCs

Tracing a Simulation

You can also use the MSC Editor as a graphical trace tool, which enables the automatic generation of an MSC from a simulation. The results of the simulation will be presented in an MSC Editor window, in which each event of interest will be appended to the chart in order to build up a chart which reflects the history of the simulation. The results of a simulation can be saved in MSC/PR form and re-read by the MSC Editor.

The commands that start up the logging of MSC events, set up the scope of trace, stop the logging of events, and so on, are given to the Simulator. See chapter 50, *Simulating a System*.

Validating a System

An MSC can be used as a means to express the requirements on an SDL system. With the MSC Editor and the SDL Explorer, a powerful validation environment is provided. The basic idea is to draw an MSC using the MSC Editor and then use the Explorer to check if there is a possible execution path for the SDL system that satisfies the MSC.

The commands that validate a system are given to the Explorer. See chapter 53, *Validating a System*.

Tracing user-defined events

The MSC editor also contains an API that enables any user or tool vendor to make interactive traces of events of their own choice (custom trace). Please see the header file `<product install dir>/include/msctrace/msctrace.h` for further details on this API.

Supported MSC Formats

The MSC Editor supports reading MSCs stored in either:

- The MSC/GR format used by the MSC Editor (in which case the default file extension is `.msc`)
- The MSC/PR format (stored on MSC/PR files, where the default file extension is `.mpr`)

By default, the MSC Editor stores diagrams in the MSC/GR format, but it can also generate MSC/PR for a diagram.

MSC/PR files can be expressed in two forms, *event-oriented*[1] or *instance-oriented*[2]. See the Z.120 recommendation for more information on these two alternative formats.

Compliance with ITU Z.120

MSC Diagrams

The MSC Editor supports virtually all of the ITU Z.120 language definition for MSCs, including most of the MSC'96 additions.

For information on the MSC support compared to Z.120, see "Compatibility with ITU MSC" on page 5 in chapter 1, *Compatibility Notes, in the Release Guide*.

1. Event oriented MSC/PR describes the MSC using the order in which the events occur, i.e. starting with the top of the diagram and downwards, providing the feeling of a global event order.
2. Instance oriented MSC/PR describes the MSC on an instance by instance basis. The feeling of a global event order within the MSC is lost.

Pagination

MSCs cannot be split into multiple pages in the same way as SDL diagrams. However, if your MSC requires more than one sheet of paper when printing, the MSC Editor takes care of splitting the chart into multiple parts to print on separate sheets of the paper size you have specified. You set up the printer pages with the Preference Manager. You can use the *Window Options* menu choice from the *View* menu to show the page boundaries on the screen.

Each part of a multiple-part chart can be printed with adjacent page markers to assist you in making up the complete chart. This feature is by default disabled and must be set as a preference or specified in the *Print* dialog.

Syntax Rules at Editing Time

Some of the MSC syntax rules are enforced during edit when you append symbols and lines to your MSC. These rules determine:

- Which symbols and lines are allowed to be interconnected
- Where symbols can be placed

This syntax checking cannot be disabled. All text is checked during editing.

Basic Operations

Starting the MSC Editor

In the Organizer, MSCs are handled as references, which you can add and remove from any of the Organizer chapters. There are several ways you can start the MSC Editor:

- *Starting the MSC Editor Stand-alone*
- *Starting on an Existing MSC Icon*
- *Starting on an Existing MSC, not Managed by the Organizer*
- *Starting on an Unconnected MSC Icon*
- *Starting on a New MSC Icon*

Starting the MSC Editor Stand-alone

You have the possibility to start the MSC Editor stand-alone. This is however only recommended if you are using the MSC Editor for viewing a generated trace since you will not be connected to the Organizer and you will loose e.g. MSC reference navigation capabilities. If you have bought the MSC as a trace tool only (i.e. only at trace license) this will be your only option.

> **Note:**
>
> If you are making a custom interactive trace, you should not start the MSC Editor/Tracer yourself, since in that case the editor will not be able to connect with the tracing application. You need only start the MSC Editor manually when viewing saved MSC diagrams or generated MSC/PR files.

On Windows you start the MSC Editor by writing **`sdtmsce.exe [ <filename> ]`** at the command prompt. It is also possible to set up a shortcut. Please refer to your operating system documentation for further information.

On UNIX you start the MSC Editor by writing **`sdtmsc [ <filename> ]`** at the command prompt.

The optional <filename> may either refer to a GR file (usual suffix .msc) or an MSC/PR file (usual suffix .mpr). If you enter an incorrect filename the MSC Editor will terminate without giving any notice.

Starting on an Existing MSC

To start the MSC Editor on an existing MSC, you either use an MSC icon connected to an MSC diagram file, or add a new MSC icon for an MSC diagram.

Starting on an Existing MSC Icon

To start the MSC Editor on an existing MSC managed by the Organizer:

1. In the Organizer select the icon that represents an MSC. (The MSC icon must be connected to a file.)
   - Alternatively, if the MSC is associated with an SDL diagram, you can select the associated MSC link icon (looking like an MSC icon displayed in dashed form).

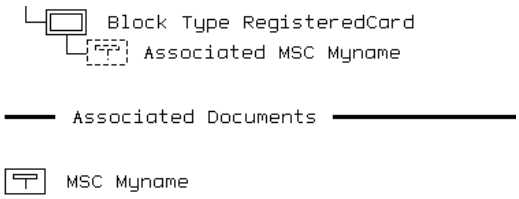


Figure 332: An MSC icon and an associated MSC icon

2. Start the MSC Editor by the *Edit* menu choice on the Organizer's *Edit* menu or by double-clicking the MSC icon.

   The MSC Editor window is displayed, the current MSC data is read from file and the upper left portion of the chart is displayed.

Starting on an Existing MSC, not Managed by the Organizer

To start the MSC Editor on an existing diagram that is not managed by the Organizer (i.e. not contained in the Organizer structure) you must first include it into the Organizer, before the MSC Editor can be started.

To include an existing MSC into the Organizer:

1. Select the place in the Organizer where you want to add the MSC. The MSC will be added as a new root diagram after a selected document or chapter, or at the top level of a selected module.
2. Select *Add Existing* from the Organizer's *Edit* menu.
3. In the dialog that is opened, find and select an MSC diagram file. Set the filename filter to `*.msc` to only list MSCs. The MSC is added to the Organizer structure, and is opened in the MSC Editor.

   If an SDL diagram was selected in the Organizer, an association link to the MSC is also added to the SDL diagram.

Starting on a New (Nonexisting) MSC

To start the MSC Editor on a new MSC, you either use an unconnected MSC icon, or add a new MSC icon.

Starting on an Unconnected MSC Icon

To start the MSC Editor on an existing but unconnected MSC icon:

1. Select an unconnected MSC icon in the Organizer.
2. Open the unconnected MSC by the *Edit* menu choice on the Organizer's *Edit* menu or by double-clicking the MSC icon. A dialog is opened:

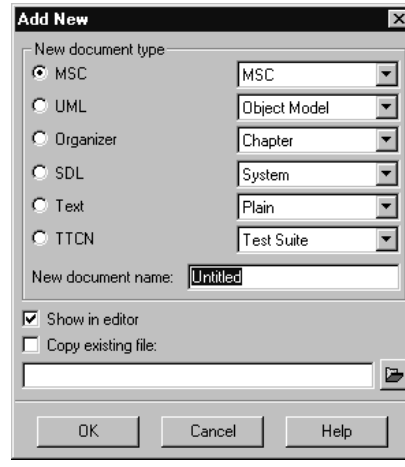


Figure 333: The Edit dialog

3. Make sure the *Show in editor* option is set. Then click *OK* to open the MSC Editor on the new, empty diagram.
   - You can also change the name of the MSC diagram before it is opened.
   - To copy the contents of an existing MSC diagram file before opening the new diagram in the MSC Editor, select the option *Copy existing file* and enter a filename.

Starting on a New MSC Icon

To add a new MSC icon to the Organizer and start the MSC Editor on that icon:

1. Select the place in the Organizer where you want to add the MSC. The MSC will be added as a new root diagram after a selected document or chapter, or at the top level of a selected module.
2. Select *Editors > MSC Editor* from the Organizer *Tools* menu. A new MSC diagram icon is added to the Organizer structure, and the new diagram is opened in the MSC Editor. If an SDL diagram was selected in the Organizer, an association link to the MSC is also added to the SDL diagram.
   - Alternatively, select *Add New* from the Organizer *Edit* menu. A dialog is opened. Select an MSC diagram type, enter a name in the *New document name* field and click *OK*.

Exiting the MSC Editor

1. Select *Exit* from the *File* menu. If the MSC Editor holds any MSC that you have modified but not saved, a dialog is issued, prompting you to save the changes before the application terminates.
2. The dialog informs you about the name of the diagram in turn to be saved. Depending on if you want to save the changes or not:
   - Click *Save* to save changes made to the current diagram. The MSC Editor prompts you to save the next modified diagram, alternatively exits if there are no more diagrams to save.
   - Click *No Save* to close the current diagram without saving it. The MSC Editor then prompts you to save the next modified diagram, alternatively exits if there are no more diagrams to save.
   - Click *Save All* to save all changes made to all diagrams. Once all diagrams are saved, the MSC Editor will exit.
   - Click *Quit All* to exit the MSC Editor without saving any MSCs.
   - Click *Cancel* if you want to continue the MSC Editor session.

Managing MSCs

You create MSCs in the MSC Editor. Once an MSC has been created, you may include it into the structure that is managed by the Organizer, in order to take full advantage of the functionality that is provided. MSCs can however be handled as separate entities.

The following operations are described in this section:

- *Creating an MSC*
- *Renaming an MSC*
- *Opening an MSC*
- *Saving an MSC*
- *Saving a Copy of an MSC*
- *Displaying an Opened MSC*.

Creating an MSC

You can create MSCs in two ways.

Creating an MSC from the MSC Editor

1. Select *New* from the *File* menu. A dialog is displayed.
2. Enter the diagram name and type and click *New*.

   The MSC is saved as a separate entity that is not automatically appended to the Organizer structure. You may want to add the MSC to the Organizer in order to provide an instant access to it from the Organizer. See "Starting on a New MSC Icon" on page 1796.

Creating an MSC from the Organizer

The procedure is identical to starting the MSC Editor with no existing MSC icon. See "Starting on an Existing MSC" on page 1793.

Renaming an MSC

You can change the name of an MSC. The name of an MSC is reflected in the MSC heading.

To change the name, simply edit the heading.

Opening an MSC

You can open an MSC from the MSC Editor or from the Organizer.

Opening an MSC File from the MSC Editor

1. Select *Open* from the *File* menu. A File Selection Dialog is issued.
2. Select a file and click *OK*.

You can open binary GR files (`*.msc`) or textual PR files (`*.mpr`).

Opening an MSC from the Organizer

1. Select an MSC in the Organizer.
   - Alternatively, if the MSC is associated with an SDL diagram, you can select the associated MSC link icon (looking like an MSC icon displayed in dashed form, see Figure 332 on page 1794).
2. Double-click the MSC icon.

Saving an MSC

There are several ways to save an MSC. Either you can save changes made to one specific MSC or all MSCs modified during the current session.

Saving an MSC in the MSC Editor

- Select *Save* from the *File* menu or click the quick-button for *Save.*

  If the diagram is newly created, a file selection dialog is issued where you can specify the file to save the diagram on.

Saving All MSCs

You can make a global save of all the MSCs that are open in the current session, and that are modified:

- Select *Save All* from the *File* menu. (You will be prompted to specify a file for new diagrams that have never been saved on file.)

Saving an MSC in MSC/PR Format

You can save an MSC in the MSC/PR format. See *"Generate MSC PR" on page 1688 in chapter 39, Using Diagram Editors*.

Saving a Copy of an MSC

You can save the MSC you are working on into another file. You may either continue working with the original file or with the newly created copy.

Continuing Working with the New Copy

1. Select *Save As* from the *File* menu. A file selection dialog is issued.

   The default file name which is suggested is the same file as the original was stored on, appended with an integer suffix (1, 2,..., N) in order to build up a unique file name.

2. Click *OK*. The MSC is copied to the specified file, the file containing the original is closed, and the window from where the save was made remains open for editing the newly created file.

Continuing Working with the Original

1. Select *Save a Copy as* from the *File* menu. A file selection dialog is issued.

   The default file name which is suggested is the same file as the original was stored on, appended with an integer suffix (1, 2,...,N) that guarantees a unique file name.

2. Click *OK*. The MSC is saved on the specified file, and the window with the file from where the save was made remains open for editing.

Displaying an Opened MSC

You can display an MSC that is currently opened by the MSC Editor. (The MSC that you would like to see may reside in a window which is not on top of the screen.)

The *Diagrams* menu shows all MSCs (up to a maximum of the last nine to have been opened). If more than nine MSCs are open, a tenth menu choice, *List All*, provides access to a list dialog where all MSCs currently opened are listed.

Rearranging an MSC

The following topics are discussed:

Resizing an MSC

Resizing Automatically

When you append objects by double-clicking, the MSC Editor will automatically enlarge the MSC size when the MSC has grown so that it reaches the bottom or right of the drawing area. The drawing area is enlarged according to the value of a preference parameter.

Resizing Manually

1. Display the MSC you want to resize.
2. From the *Edit* menu, select *Drawing Size*. A dialog appears.
3. Type in the new width and height for the MSC within the allowed range.
4. Click *OK*. The MSC is resized accordingly.

Changing the Spacing

When the MSC Editor layouts an MSC, it makes use of a horizontal and vertical spacing parameter that affect the total height and width of the MSC. All instances and events on instances are positioned with an integer multiple of the spacing parameters.

1. From the *View* menu, select *Insert Options* (see *"Insert Options" on page 1677 in chapter 39, Using Diagram Editors*). A dialog appears.
2. Drag the sliders for a coarse adjustment. Click left or right on the slider bar for a fine adjustment.
   - Smaller values will give the chart a compact appearance that may possibly make it more difficult to read. Text associated to

symbols may overlap other symbols and text. Symbols may overlap each other.

– Larger values will give the chart a less cramped appearance and makes it easier to read to the price of a larger size.

3. Click *OK*.

Managing Windows

Each MSC Editor window shows one MSC, but you may open new windows on a diagram and close windows that are no longer needed.

A newly opened window is a mirror image of the window from which the selection to open a window was made. Multiple windows may be opened and are all identical to the first. Any subsequent modifications made will reflect the same in either the original window, or any of the newly opened window(s). This option affords the possibility of viewing in more than one window, and can ideally be used in conjunction with the scaling factor when looking at a detailed MSC.

Opening and Closing Windows

You open and close windows with the *New Window* and *Close Window* menu choices in the *Windows* menu. If you close the last opened window, the editor will exit.

Hiding and Showing Parts of the Window

The component parts of the MSC Editor window can be shown or hidden as required. **On UNIX**, if they are hidden, it increases the drawing area available to be used in the creation or modification of a diagram.

To change the options for hiding and showing parts of the window, select *Window Options* from the *View* menu. These options can also be set as preferences, and some of them are available from quick-buttons:

- The quick buttons for *text window on / off* or *symbol menu on / off*.

- The quick button for *instance ruler on / off*.

Selecting Objects

To select an object, you click it. The object will be marked with selection squares that indicates that it is selected. A filled selection square (not a grey) means that you can resize or move the object by dragging the square.

Related topics are:

- *Selecting an Object That Has Associated Objects*
- *Extending, Reducing or Cancelling a Selection*
- *Requesting Detailed Information on an Object*.

Selecting an Object That Has Associated Objects

An object may have other objects associated to it. When you select an object, all of the object's attributes are also selected:

- A **symbol** may have associated **lines** and **text** attributes.
- A **line** may have associated **text** attributes.
- A **text** object has **no** associated attributes.

Associated text, for example symbol text, symbol names and timer names, will be displayed in the text window.

When you open a condition or MSC reference symbol, the *Connect* dialog is also opened.

Extending, Reducing or Cancelling a Selection

The editor provides standard functions for extending, reducing and cancelling selection of objects.

Requesting Detailed Information on an Object

The *info window* is an auxiliary window that provides additional information about the object you have selected.

To open an info window:

1. Display the diagram on which you want additional information.
2. Select *Info Window* from the *Window* menu. The info window is displayed.

Adding and Removing Objects

The topics discussed in this section are:

Adding and Placing Symbols

1. You can add a symbol from the *symbol menu* to a chart in two ways:

   – **Automatically**, by double-clicking the symbol in the symbol menu. The symbol will be placed at a location defined according to the type of symbol and in relation to the current selection. This will be further described for each symbol type.

   – **Manually**, by first selecting it in the symbol menu and then clicking at the desired location in the drawing area. To cancel the operation before the symbol is added, press `<Esc>`.

2. Enter the symbol's text attributes.

Adjusting Objects to the Grid

The MSC Editor positions symbols and lines automatically. When you draw and move symbols and lines, each object is positioned on the closest intersection in the grid.

Inserting Space for Events

When you work on an MSC, the space available between two events on an instance axis may not be sufficient for inserting new objects on the instance axis. The MSC Editor can reorganize the MSC in order to provide the required space.

1. Select the object before or after where to insert space.

2. From the *Edit* menu, select *Make Space*. Alternatively click the quick-button for *Make space for new events*. The *Make Space* dialog is issued.
3. Turn the appropriate radio button on, depending on if you have selected the object before or after where to insert space.
4. Type in the required number of events to insert (the default is 1 event). A negative number implies removal of space.
5. Click *OK*. The MSC is rearranged and space is created on the instance axis/axes.

Inserting Space for Events Automatically

Space is made on the instance axis when you select and add new symbols from the symbol menu or when you move symbols in the diagram.

1. Select a symbol in the symbol menu and move the cursor to the place on the instance axis where you want to insert the symbol.
2. Click the mouse. The MSC is rearranged and space is created on the instance axis/axes.

Removing Space between Events

1. Select the object before where to remove space.

2. Click the quick-button for *Remove space between events*.

Removing Objects

You can remove any selected object(s) this way:

1. Select the objects to remove.
2. Choose *Clear* from the *Edit* menu. The objects are removed.

> **Note:**
> Selecting an object to clear may implicitly select other objects. The *Clear* command will delete them all. See Figure 334.

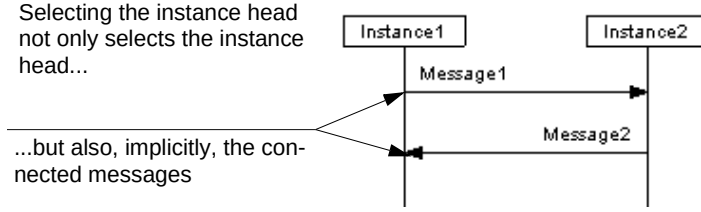


Figure 334: Selecting an object that has associated objects

Clearing the instance Instance_1 will also remove the messages Message 1 and Message 2.

Adding a Text Symbol

To add a text symbol automatically, double-click the text symbol in the symbol menu. The text symbol will be placed at the first available position at the top of the drawing area, starting from the left of the diagram.

Adding a Comment Symbol

The following conditions apply to comment symbols in the MSC Editor:

- A comment symbol can only be connected to **one** symbol or event.
- Only one comment symbol may be connected to an object.
- A comment symbol cannot be connected to another comment symbol or a text symbol.

To add a comment symbol automatically:

1. Select the object to which you wish to connect the comment.
2. Double-click the comment symbol in the symbol menu. The comment symbol will be placed to the right of the selected object and be connected to it. (If no object was selected, the comment symbol will be placed at an empty position, and you will have to connect the symbol to an object manually, as described next.)

To add a comment symbol manually:

1. Select the comment symbol in the symbol menu and place it in the drawing area.
2. Click the handle on the comment symbol, and then click on the object you wish to comment. A dashed line is drawn between the objects.

Adding an Instance Head Symbol

You can add a new instance either directly by using the instance head symbol, or you can visualize an instance that is started dynamically by using the *process create* line. The latter approach is described in "Drawing a Process Create" on page 1816.

To add an instance head symbol automatically:

1. Select, if you wish, an object to use as reference point when positioning the instance head.
2. Double-click the instance head symbol in the symbol menu.
   - If an object was selected in the drawing area, the instance head symbol will be placed to the right of the selected object.
   - If no object was selected, the instance head symbol will be placed on top of the MSC, at the first available location starting from the left.
3. Fill in the instance head text fields. See Figure 335.

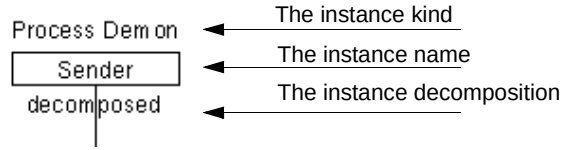


Figure 335: The instance head text fields

Adding an Instance End Symbol

The instance end symbol determines the end of the description of an instance within an MSC. It does not describe the actual termination of the instance. That is done with the *stop* symbol, see "Adding a Stop Symbol" on page 1808.

If you position the instance end symbol above (before) a symbol or line on the instance axis, all symbols and lines below (after) that object will be deleted. This is also true when you add a stop symbol:

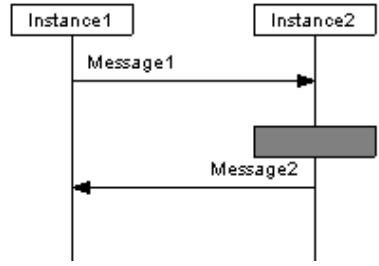


Figure 336: Placing an Instance End

The placement of the instance end symbol deletes message Message2 but not Message1.

To add an instance end symbol automatically:

1. Select an object on the instance axis immediately above (before) the desired endpoint.
2. Double-click the instance end symbol in the symbol menu.

All symbols and lines on the instance axis below it are deleted. The instance axis line is truncated and connected to the instance end symbol.

Adding a Stop Symbol

The *stop* symbol at the end of an instance body indicates the termination of that instance, as opposed to the *instance end* symbol, which only determines the end of the description of the instance within the MSC.

If you position the stop symbol above ("before") a symbol or line on the instance axis, all symbols and lines below ("after") that object will be deleted. See Figure 336 on page 1807.

To add a stop symbol automatically:

1. Select an object on the instance axis immediately above (before) the desired endpoint.
2. Double-click the stop symbol in the symbol menu.
3. The instance axis line is truncated and is connected to the stop symbol.

Drawing a Message

To draw a message you need to know from which instance it should be issued (the *source instance)* and to which instance it should be sent (the *target instance*).

Drawing a Message Automatically

1. Select the object on the source instance axis immediately above (before) the place where you wish to draw the line.
2. Double-click the message symbol in the symbol menu. The MSC Editor inserts the base end of the message on the instance axis, with the target end of the message (the arrow) marked as unconnected (lost) and facing right. (Figure 337)

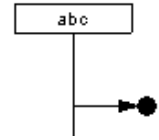


Figure 337: A lost message

3. Drag the target end to (or close to) the target instance axis and release the mouse button to connect the line. If you do not connect the end of the message, it will remain unconnected (lost), indicated by the filled circle.

4. Enter the message name and its parameters.

Drawing a Message Manually

1. Click the message symbol in the symbol menu.

2. To create a **message** or **lost message**:

   Point at the source instance axis and click once (the circle indicating a found message disappears when you are close enough to an instance axis). From now on, the base of the message is fixed to the source instance. The message is displayed as lost, following the mouse motion. (See Figure 337.)

   Or

   To create a **found message**:

   Point between two instance axes and click once. The base of the message is fixed at this position.

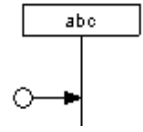


Figure 338: A found message with a connected end

3. Point at or close to the target instance axis, and click to connect the message end to the target instance. The line is connected pushing down subsequent symbols on the instance axis when necessary.

If you do not connect the end of the message to an instance axis, it will remain unconnected (lost), indicated by the filled circle.

4. Enter the message name and message parameters.

Inserting the Message Name and Message Parameters

1. The message name can be entered immediately.
2. Click the parameter selection rectangle (the lower one) so that it is the only selected object. Type the parameter list. The parenthesis around the parameter list are not displayed on the screen until you have entered a parameter.

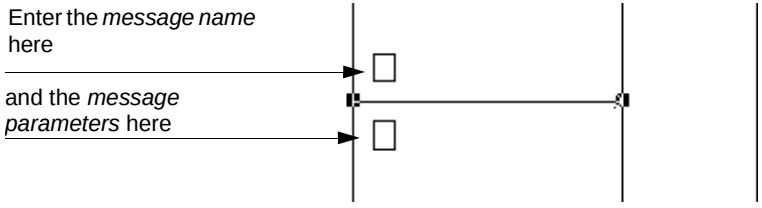
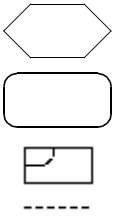


Figure 339: The message text fields

Drawing a Message-to-self

Drawing a message-to-self is done similarly to drawing a message. (See "Drawing a Message" on page 1808.) The source instance is however always the same as the target instance.



Adding a Condition, MSC Reference or Inline Expression Symbol

Adding a Condition, MSC Reference or Inline Expression Symbol Automatically

1. Select the object immediately above (before) the place on the instance axis where you wish to insert the symbol.
2. Double-click the condition, MSC reference or inline expression symbol in the symbol menu.
3. Type the condition or MSC reference name, or fill in the inline expression text.

4. Connect the condition to the instances. See "Connecting the Condition or MSC Reference Symbol" on page 1811.

Adding a Condition, MSC Reference or Inline Expression Symbol Manually

1. Select the condition, MSC reference or inline expression symbol in the symbol menu.
2. Position the symbol at the desired point of the instance axis. The symbol is placed on the instance axis, pushing down subsequent symbols on the instance axis when necessary.
3. Type the condition or MSC reference name.
4. Connect the symbol to the instances. See "Connecting the Condition or MSC Reference Symbol" on page 1811.

Connecting the Condition or MSC Reference Symbol

In the MSC Editor you can connect the condition or MSC reference symbol to individual instances or perform a global connection.

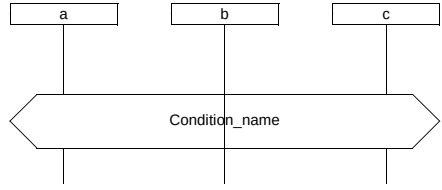


Figure 340: A condition symbol

When a symbol straddles an instance axis to which it is not connected, the axis line is drawn through the symbol. In Figure 340 the condition symbol with name Condition_name is connected to the instances a and c, but not to the instance b.

To connect a condition or MSC reference to a number of instance axes:

1. Select the symbol to connect.
2. Select *Connect* from the *Edit* menu. The *Connect* dialog is opened:

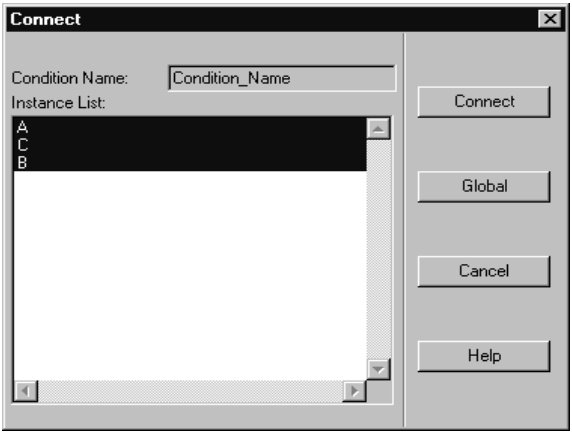
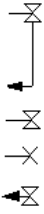


Figure 341: Connecting an instance

3. Select the instances to be connected in the instance list. The instances that will be connected are highlighted.
   - Click *Connect* to connect the condition or MSC reference symbol to the instances you have selected.
   - Click *Global* to connect the condition or MSC reference symbol globally to all instances in the list.



Drawing a Timer

Drawing a *timer* is done similarly to drawing a message line. The *source* instance is always the same as the *target* instance.

Drawing a Timer Automatically

1. Select the object on the instance axis immediately above (before) the place where you wish to draw the timer.
2. Double-click the timer symbol in the symbol menu. The MSC Editor inserts the timer.
3. Drag the target end and release the mouse when you have reached the point where the timer should expire.

4. If required, change the status of the timer by selecting *Status* from the *Edit* menu (see "Changing the Status of a Timer" on page 1814).
5. Enter the timer name and its parameters.

Drawing a Timer Manually

1. Click the timer symbol in the symbol menu.
2. Point at or close to the instance axis. Click once. The base of the timer is fixed and the timer is displayed as a vertical line, following the mouse motion.
3. Point at or close to the point on the axis where the timer should expire and click to connect the *timer end*. The timer end is connected and the timer is assigned the status **timeout**.
4. Enter the timer name and parameters.

Drawing a Separate Timer Manually

1. Draw a timer.
2. Select one of the outermost selection squares (i.e. the selection squares **not** on the instance axis) and drag it so that the vertical extension of the timer disappears.
3. Enter the timer name and parameters.

Changing Between Timer and Separate Timer

- Change a timer to a separate timer by selecting one of the outermost selection squares (i.e. the selection squares **not** on the instance axis) and dragging it so that the vertical extension of the timer disappears.
- Change a separate timer to a timer by selecting a selection square and dragging it vertically.

Inserting the Timer Name and Timer Parameters

1. The timer name can be entered immediately.
2. Click the parameter selection rectangle (the lower one) so that it is the only selected object. Type the timer parameters. The parenthesis around the timer parameters are not displayed on the screen until you have entered a parameter.

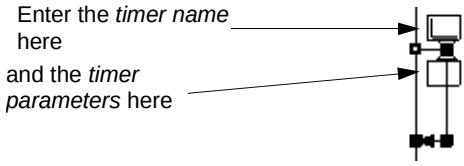


Figure 342: The timer text fields

Changing the Status of a Timer

Z.120 specifies two appearances for the timer symbol, depending on whether the timer has expired (i.e. timeout) or whether the timer is reset.

- **Timeout** – A timer expires.
- **Reset** – A timer is stopped.
- **Implicit reset** – A timer is assigned a new value while active.To change the status of a timer:

1. Select the timer.
2. From the *Edit* menu, select *Status*. The *Timer status* dialog is issued.

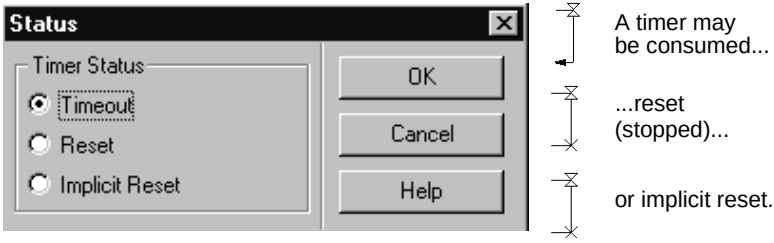


Figure 343: Setting the timer status

3. Click a radio button and then click *OK*. The status of the timer is changed accordingly.

Changing the Status of a Separate Timer

The status of a separate timer:

- **Set** – A timer is activated but has not yet expired.
- **Timeout** – A timer expires.

- **Reset** – A timer is stopped.

To change the status of a timer:

1. Select the timer.
2. From the *Edit* menu, select *Status*. The *Timer status* dialog is issued.

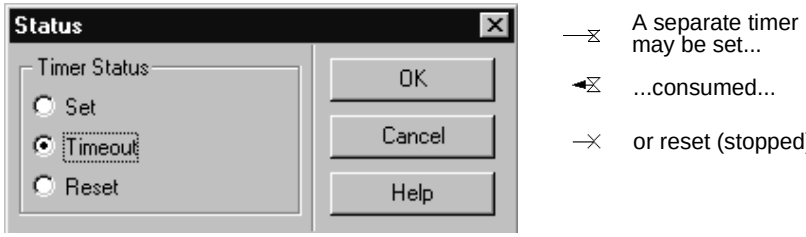


Figure 344: Separate Timer status

3. Click a radio button and then click *OK*. The status of the timer is changed accordingly.

Adding an Action Symbol

An *action* symbol corresponds to the SDL *task* symbol. You may choose to fill it with formal or informal expressions.

To add an action symbol automatically:

1. Select the object on the desired instance axis immediately above (before) the place where you wish to insert the symbol.
2. Double-click the action symbol in the symbol menu. The symbol is placed on the instance axis, pushing down subsequent symbols when necessary.
3. Fill in the action text.

Drawing a Process Create

Drawing a Process Create Automatically

1. Select the object on the **source** instance axis immediately above (before) the place where you wish to draw the process create.
2. Double-click the *create process* line in the symbol menu. The editor draws the create process line and places the instance head to the right of the source instance axis.
3. Enter the create parameters and the created instance head text fields (see Figure 335 on page 1807).

Drawing a Process Create Line Manually

1. Click the process create symbol in the symbol menu.
2. Click once on the **source** instance axis. The base of the process create line is fixed to the source instance. The line is displayed as a dashed line.
3. Click at the location where you wish to place the head of the created instance. The instance head and instance axis are automatically drawn.
4. Enter the create parameters and the created instance head text fields.

Adding a Coregion Symbol

The coregion allows you to specify unordered events on an instance. A coregion covers for instance the practically important case of two or more incoming messages where the ordering of consumption may be interchanged.

1. Select the coregion symbol in the symbol menu.
2. Click once to designate the start point of the symbol at the instance axis.
3. Click a second time on the same instance axis to designate the end point.

Collapsing and Decomposing Diagrams

You can create new high level and low level diagrams of your original diagrams. The topics discussed are:

Collapsing a Diagram

You can collapse the selected instance axes in a diagram to get an overview of the diagram.

1. Select two or more instance axes.
2. Select *Collapse* from the *Edit* menu.

   Two diagrams are created from the original diagram. One displaying the collapsed instances and one displaying the collapsed diagram.

See "Expand/Collapse" on page 1671 in chapter 39, *Using Diagram Editors* for more information.

Decomposing a Diagram

You can create a new diagram of a segment of the original diagram.

1. Select an instance axis.
2. Select *Decompose* from the *Edit* menu. A dialog is opened.
3. Enter or change the name of the decomposed instance and click *OK*.

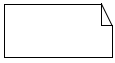
   The instance in the original diagram will be marked as decomposed. A decomposed diagram is created and it is displayed in the Organizer.

See "Decompose" on page 1675 in chapter 39, *Using Diagram Editors* for more information.

Moving Objects

You can move one or several objects on the chart. Move several objects by first selecting them by framing or by using the shift buttons. The various procedures are described below:

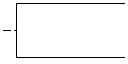
- *Moving a Text Symbol*
- *Moving a Comment Symbol*
- *Moving an Instance Head Symbol*
- *Moving an Instance End Symbol and a Stop Symbol*
- *Moving an Instance Axis*
- *Moving a Message*
- *Moving a Message-to-self*
- *Moving a Timer*
- *Moving a Process Create*
- *Moving a Condition or MSC Reference*
- *Moving an Inline Expression Symbol*
- *Moving an Action Symbol*
- *Moving a Coregion Symbol*.



Moving a Text Symbol

You can move the *text* symbol freely across the drawing area.

- Drag the symbol to the new location.



Moving a Comment Symbol

You can move the *comment* symbol freely across the drawing area.

- Drag the symbol to the new location. The symbol and the line are redrawn accordingly.

  The orientation of the symbol may change and the connecting line is redrawn as required:

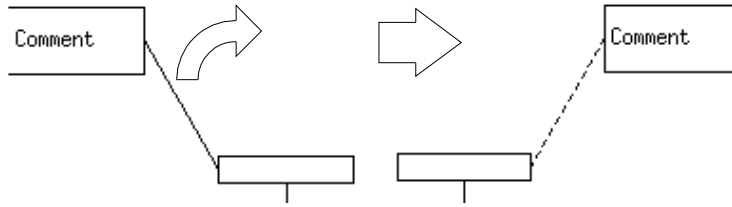


Figure 345: Moving the comment symbol.

Moving the comment symbol as shown in Figure 345 mirror flips the symbol in order to provide the shortest possible connection line.

If you move the object that the comment symbol is connected to, the comment symbol moves with it.

Moving an Instance Head Symbol

The *instance head* symbol can be moved horizontally or vertically.

- Drag the instance head to the new location.
  - If you induce a horizontal movement, the entire instance axis and the connected objects are redrawn accordingly.
  - If you induce a vertical movement, only the instance head is moved. The objects on the instance axis are not affected.

Restrictions

The following restrictions apply when moving an instance head vertically:

- You can move the instance head symbol upwards until it reaches the limit of the drawing area. See (1) in Figure 346 on page 1820. No other symbols or lines are affected by this operation.
  - An instance head symbol that belongs to a process create line cannot, however, be moved above the head of its parent instance axis.
- You can move the instance head symbol downwards until it reaches the *instance end symbol* or until it reaches the first *event (condition, action, message, timer, stop* or *process create*) that is connected to

its *instance axis*. No other symbols or lines are affected by this operation. See (2) in Figure 346 on page 1820

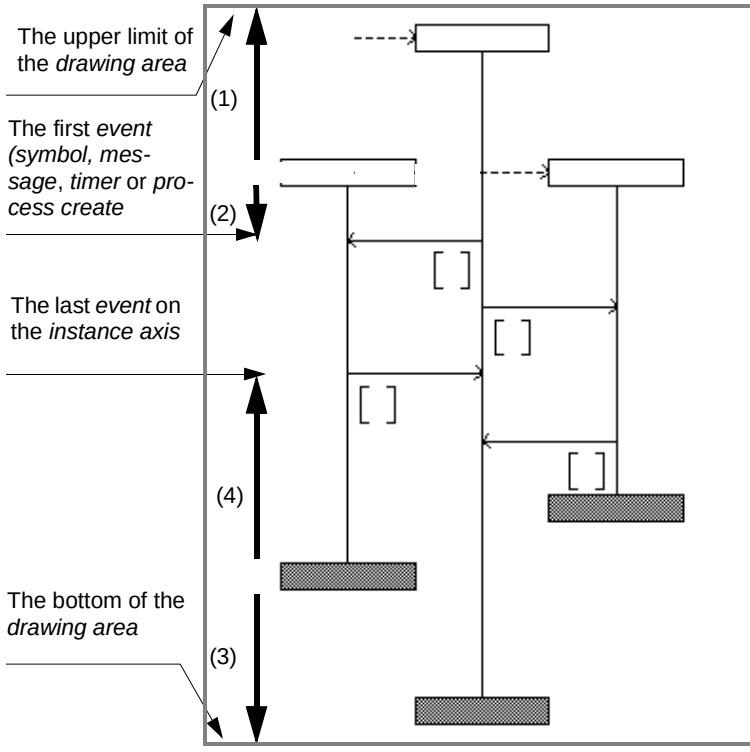
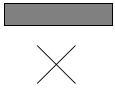


Figure 346: Moving the instance head and the instance end symbols

The bold arrows show the limitations applied on the leftmost instance head and instance end.



Moving an Instance End Symbol and a Stop Symbol

You can move the instance end symbol and stop symbol vertically but not horizontally. The instance end is moved horizontally when you move the instance head or the instance axis horizontally.

You can move the symbol vertically:

- Down until it reaches the bottom of the drawing area. See (3) in Figure 346.
- Up until it reaches the last event (action, condition, message, timer or process create) that is connected to the instance axis (or reaches the instance head symbol). See (4) in Figure 346 on page 1820.

Moving an Instance Axis

You can move the instance axis line left or right by dragging it with the mouse. All connected messages and process create lines remain connected as they were initially. The instance head and instance end or stop symbols will be moved accordingly. See (1) and (2) in Figure 347.

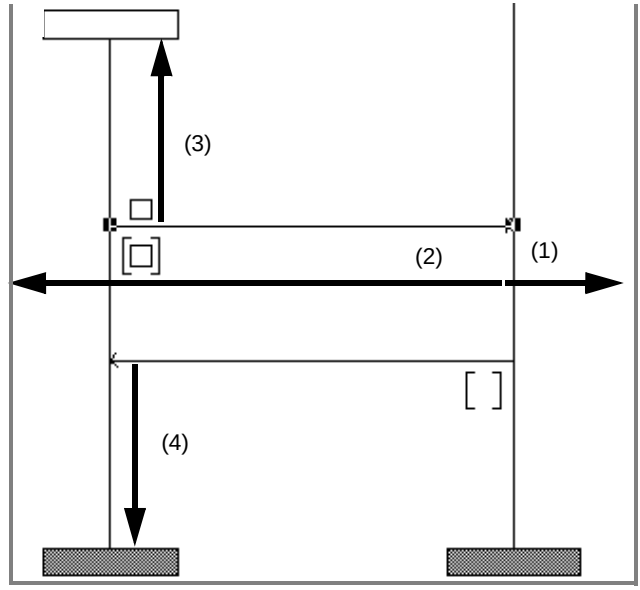


Figure 347: Moving lines

To move an instance axis:

- Drag the instance to the new location. The entire instance axis and the connected objects are redrawn accordingly.

Moving a Message

You can move a message vertically to any position within the limits set by the instance head symbol and the instance end or stop symbol of the source and target instances.

Moving the Entire Message

The base and the end of the message remain connected to the original source and target instances. The related message name and message parameters will be moved accordingly. See (3) and (4) in Figure 347.

You can also move the entire message to another pair of instance axes.

To move a message:

- Drag the message to the desired position. The message and its text attributes are redrawn accordingly.

Moving the Base or End of a Message

You can move the base or end of a message up or down along the axis that it is connected to by pointing to and dragging the appropriate (base or end) selection square to the new position. The opposite selection square will remain fixed.

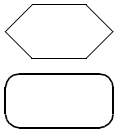
The end (message input) cannot be moved above the base (message output).

Moving a Message-to-self

You can move a message-to-self up or down along its instance axis or between instance axes by dragging it as you would move a message. See "Moving a Message" on page 1822.

Moving the Base or End of a Message-to-Self

Use the same procedure as for a message to move the end (message input) or base point (message output) of a message-to-self (see "Moving the Base or End of a Message" on page 1822).



Moving a Condition or MSC Reference

You can move the *condition* and *MSC reference* symbol vertically along an *instance axis* to which it is connected; between the instance head and instance end, stop, or the bottom of the drawing area. To connect or disconnect instances to the symbol, use the *Connect* dialog on the *Edit* menu.

To move the condition and MSC reference symbol:

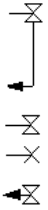
- Drag the symbol to the new location on the instance axis. The symbol is redrawn at its new location.

Moving an Inline Expression Symbol

You can move the *inline expression symbol* vertically along the instance axis.

- Click the left or right side of the inline expression and move it.

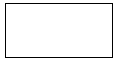
If the inline expression contains an involved axis, the inline expression cannot be moved below the instance end or stop symbol of that axis.



Moving a Timer

You can move a timer up or down along its instance axis or between instance axes by dragging it as you would move a message. The related timer name and parameters will be moved accordingly. See "Moving a Message" on page 1822.

Use the same procedure as for a message to move the end (the time-out or reset (stopped) arrow) or base point (the set symbol) of a timer (see "Moving the Base or End of a Message" on page 1822).



Moving an Action Symbol

You can move the *action* symbol vertically along the instance axis between the instance head and instance end, stop or the bottom of the drawing area or to another instance axis.

Moving a Process Create

To move a process create line along its source instance axis, either move the process create line or its associated instance head symbol upwards or downwards. See "Moving an Instance Head Symbol" on page 1819.

Moving a Coregion Symbol

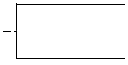
You can move a *coregion* up or down along its instance axis by dragging it as you would move a message. See "Moving a Message" on page 1822.

If you point close to the start or end point of the coregion and start dragging the mouse, the coregion will be resized accordingly.

Reconnecting Objects

Some of the objects that are managed by the MSC Editor can be reconnected:

- *Reconnecting a Comment Symbol*
- *Reconnecting a Condition or MSC Reference Symbol*
- *Reconnecting a Message*
- *Reconnecting a Message-to-self*
- *Redirecting a Message*.
- *Reconnecting a Process Create*



Reconnecting a Comment Symbol

A *comment* symbol can be reconnected to a different object or disconnected completely from all objects.

1. Drag the comment symbol connection handle towards the new connection point. The existing connection line is erased and a new line that follows the position of the mouse pointer is drawn.
2. Click the connection point on the new object. The connection line is fixed to the new object.

Disconnecting the Comment Symbol

- Select the dashed line to the comment symbol and delete it.

Reconnecting a Message

You can reconnect the base or the end of a message to another instance axis:

1. Drag the base or end of the message towards the desired instance axis. As soon as you move the mouse the message is unconnected.
2. When the mouse pointer is close to the new instance axis, release the mouse button. The message is now reconnected.

Reconnecting a Message-to-self

You can reconnect the base or end of a message-to-self to another instance axis. This changes the message-to-self to an ordinary message.

- Drag the end (or the base) of the message-to-self away from the instance axis. As soon as you move the mouse, the line is displayed as an ordinary message line. Continue dragging the end of the line up or down to the new connection point and release the mouse button.

Redirecting a Message

When you draw a message, the message goes from the source instance to the target instance. It is possible to redirect a message (i.e. change its direction):

1. Select the message to redirect.
2. Select the *Redirect* command on the *Edit* menu. The message is redrawn in the opposite direction, and the text attributes are positioned accordingly.

Messages-to-self and *overtaken messages* cannot be redirected.

In Figure 348 two messages are shown. The message sent first is consumed after the message sent last, i.e. it is overtaken.

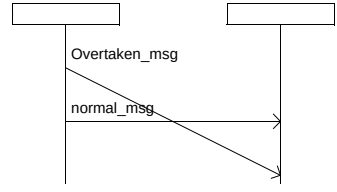
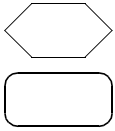


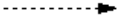
Figure 348: An overtaken message



Reconnecting a Condition or MSC Reference Symbol

To connect or disconnect instances to the symbol:

- Use the *Connect* dialog on the *Edit* menu, see *"Connect" on page 1674 in chapter 39, Using Diagram Editors*. (Remember, at least one instance must be left connected.)



Reconnecting a Process Create

You can reconnect the *base* of a *process create* from one instance axis to another in a similar way as when reconnecting a message (see "Reconnecting a Message" on page 1825).

Resizing Objects

All objects that are managed by the MSC Editor are automatically resized according to the size of the text. However, you can manually resize a text or comment symbol.

- *Resizing a Text or Comment Symbol*

Resizing a Text or Comment Symbol

There are two methods of resizing the text and the comment symbol:

- Select the *Expand/Collapse* command on the *Edit* menu to toggle between maximum and minimum size.
- Double-click the symbol, which causes the symbol to toggle between its minimal and maximal size.

Printing Objects

It is possible to print only selected parts of an MSC.

1. Select the objects to be printed.
2. Select *Print* from the *File* menu.
3. Click the *Setup* button.
4. Select the option *Print only selected symbols*.

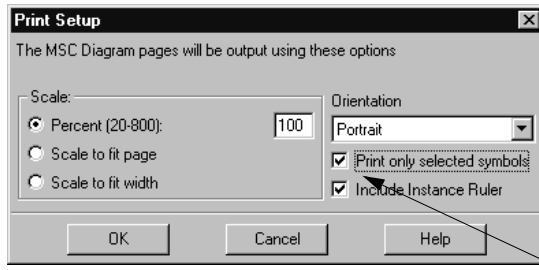


Figure 349: Printing only selected objects

5. Adjust, if necessary other *Print options* and click the *OK* and *Print* buttons.

See also "Printing from the MSC Editor" on page 312 in chapter 5, *Printing Documents and Diagrams*.

Chapter 41 Editing MSC Diagrams

Chapter

42 *The UML2SDL Utility*

The UML2SDL utility converts a model described in UML to an SDL system. This includes conversion of UML Static Structure diagrams to corresponding SDL concepts, as well as translation of UML (Harel) State Chart diagrams to SDL process diagrams.

This chapter includes setup instructions and a description of the functionality of the tool. In the end of the chapter, the mapping rules are described.

This guide assumes that you are familiar with the concepts of UML – static structure diagrams – as well as SDL.

Setting Up the UML2SDL Utility

To efficiently run the UML2SDL utility, you will need to set up a specialized Organizer menu bar containing the menu *UML To SDL*. This modified Organizer menu is defined in the file `org-menus.ini` that is located in `/orca/uml2sdl/examples/` in the installation directory. How to add the UML2SDL menu to the Organizer is described in "Defining Menus in the SDL Suite" on page 18 in chapter 1, *User Interface and Basic Operations*.

Converting UML Diagrams

When UML diagrams are to be converted into SDL diagrams, the UML diagrams need to be placed within a module in the Organizer. The diagrams may be either static structure diagrams or state charts. If multiple diagrams exist within the module, all diagrams will be converted at the same time.

To convert UML diagrams in a module:

1. Select a diagram within the module.
2. Select the desired conversion alternative from the *UML To SDL* menu (see below).

The UML2SDL converter will create a new module with the same name as the converted module, but with the prefix "SDL_" added. The new module will contain all the resulting SDL diagrams.

The *UML To SDL* Menu

The *UML To SDL* menu in the Organizer contains four alternatives:

- *Generate SDL System* – generates a system using the default transformation options.
- *Generate SDL Package* – generates a package using the default transformation options.
- *Generate SDL System* – allows you to configure the generation of an SDL system using transformation options.
- *Generate SDL Package* – allows you to configure the generation of an SDL package using transformation options.

Transformation Options

The UML2SDL utility is run as a command-line tool, but is started from the Organizer. If you select an alternative in the *UML To SDL* menu that allows you to change the transformation options, a dialog is opened in which you may specify the transformation options to UML2SDL:

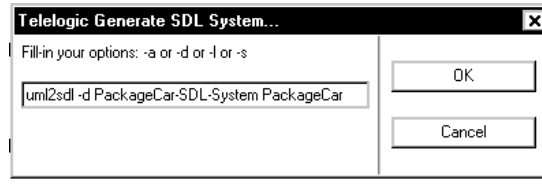


Figure 350: Setting transformation options for a package generation

If you select an alternative in the *UML To SDL* menu that uses the default transformation options, the dialog is not opened and none of the options described below are used.

The UML2SDL utility accepts a set of flags which allow you to configure the transformation:

```
uml2sdl [ -a | -d | -l | -o | -p | -s ] <module>
```

- `-a`
  Avoid types: All *«process»* and *«block»* classes that can have a type property, will automatically have this type set to “false”.
- `-d <directory>`
  The name of the sub directory that will contain the generated SDL files. A suggestion is given in the dialog, which you can alter. If the directory name does not match an existing sub directory, a new one will be created inside the current working directory.
- `-l`
  Local types: Push all type definitions as low as possible in SDL block hierarchies.
- `-o`
  Output SDL/PR to stdout, that is, the generated SDL/PR will be output in the Organizer log. No SDL diagram is created in the Organizer.

- `-p`
  If used, the UML package will be translated to an SDL package; otherwise it will be translated to an SDL system. This option is preset on the command line if you selected *Generate SDL Package* from the *UML To SDL* menu.
- `-s`
  Signal default: All operations with no return values are considered to be signals and not remote procedures. If not used, only operations following the *«signal»* stereotype or operations with a property "{async}" will be mapped to signals.
- *<module>*
  The name of the package or system that will be created. A suggestion is given in the dialog, which you can alter.

Transformation Rules

The transformation rules for UML static structure diagrams are described below. For information about the transformation rules applied when converting state charts, please refer to *"Converting State Charts to SDL" on page 1702 in chapter 39, Using Diagram Editors*.

General

To ensure traceability between the UML model and the SDL model, all mapped entities keep their name throughout the models.

An Organizer module is considered to represent a UML package. A module may contain several UML static structure diagrams. The module, including all UML static structure diagrams, is transformed into an SDL package or system.

Each SDL system or package will also contain a block type representing the architecture of the UML model.

Classes

The UML2SDL utility makes some basic assumptions about how a class should be interpreted in the SDL context. The basic approach is however to instruct the tool by using stereotypes. A stereotype is a meta classification of a class. The tool relies on the following stereotypes:

- A class with the stereotype «newtype» becomes an SDL newtype.
- A class with the stereotype «process» becomes an SDL process.
- A class with the stereotype «block» becomes an SDL block.

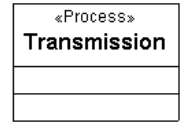


Figure 351: A class with the stereotype « process »

A class may have attributes and operations. Depending on how the class is interpreted, attributes become variables in a process, or fields in a struct. An attribute have a name and a type; the type may be omitted. The notation for the attribute is:

```
< name > [ ':' < type >]
```

An operation has a name. It may also contain parameters and return values. An operation with a return value is translated to a remote procedure; otherwise it is translated to a signal. The notation of an operation is:

```
< name > ['(' { < parameter > [':' < type > ] }*
')' ] [':' < return value >]
```

A UML static structure diagram may contain references to classes defined in other packages. Such classes are given the name according to the following notation:

```
< Name of external package > '::' < class name >
```

Each externally defined class will generate a use clause, referring to the external package, in the generated SDL system or package.

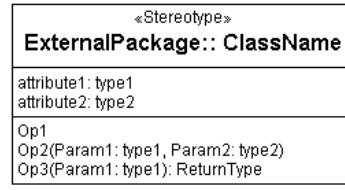


Figure 352: The notation for a process

Relations

Inheritance

Two classes connected with an inheritance relationship will generate an inheritance clause in the subtype. If two «process» classes are related by the inheritance relationship, there will be an inheritance clause in the process type representing the subclass.

Inheritance relationships between two «newtype» classes are not allowed due to limitations in SDL.

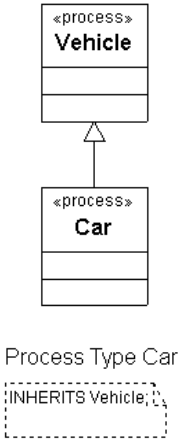


Figure 353: An inheritance relationship in the UML model will result in an inheritance relationship in the SDL model

Aggregation

Two classes may be related by an aggregation. Depending on the context the following translations will be made:

An aggregation between a «block» class and a «process» class will place the process inside the generated block type:

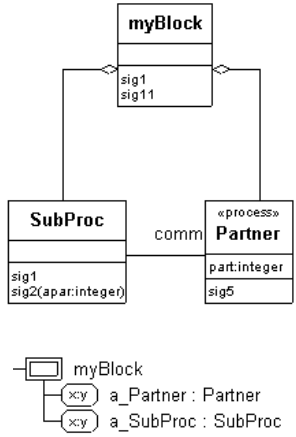


Figure 354: Aggregation expressed in UML and the generated SDL diagrams in the Organizer view

An aggregation between a «process» class and a «newtype» class will place a variable of the newtype in the process. The newtype definition will be placed in the block containing the process.

Association

Two classes connected with an association will be transformed as two classes and connected with a signal-route and/or a channel depending on the context.

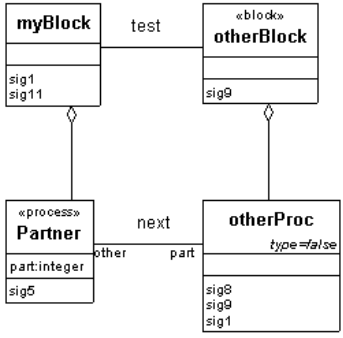


Figure 355: Communicating classes

In Figure 355 above, two classes “Partner” and “otherProc” are connected by an association. In the generated SDL the two processes “otherProc” and “Partner” will exist in two separate blocks and they will be able to communicate through a channel and two signal routes.

State Charts

State Charts placed inside the module will be converted together with the other diagrams inside that module. The converter assumes that if a state chart has the same name as a process then the converter will try to merge the resulting SDL process behavior into the diagram of that process.

A Small Example

This example is intended to show how the UML2SDL utility can be used. The example contains a small analysis model of a game – the Demon game – which is intended to be implemented in SDL through a design model. The Demon game is used as an example in other parts of the documentation. For example, see "The Demon Game" on page 41 in chapter 3, *Tutorial: The Editors and the Analyzer, in the SDL Suite 6.2 Getting Started*.

Model Relationships

The purpose of an analysis model is to identify the problem: **what** is to be done? The following model is the design model which identifies the solution. The design model answers the question **how** it is to be done. Good practice is to have clear dependencies between the two models, that is, traceability. Traceability is one of the key factors in a successful project.

Very often, the analysis model can be reused when the design model is created. The information provided by the problem statement is needed in the design model. The purpose of the UML2SDL utility is to automate this reuse as much as possible.

The Analysis Model

The Class Diagram

The analysis model is presented in Figure 356. It contains a set of classes which describes an overview of the Demon game. A pure analysis model might not be as detailed as the one given in the example, but the level of detail in the example is chosen to highlight the functionality of the UML2SDL utility.

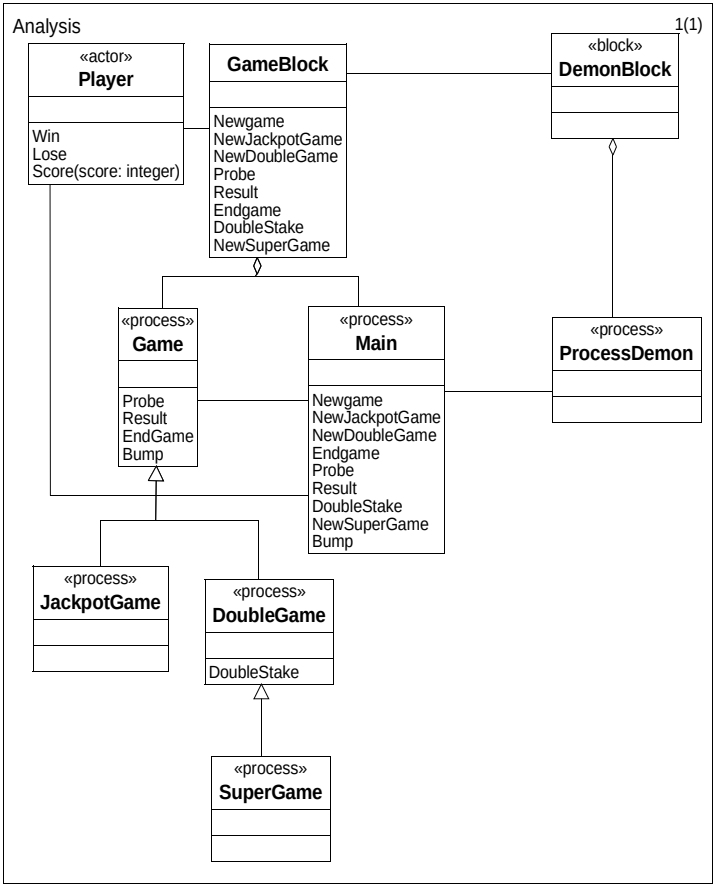


Figure 356: The Analysis Object Model

All operations of the classes in the Analysis Object Model will be mapped to signals since no operation contains a return value. Aggregate classes containing «process» classes, or marked with the stereotype «block», like GameBlock and DemonBlock, will become SDL blocks. The aggregations also tell the UML2SDL utility where to place the «process» classes. For example, the process Demon will be placed inside the block DemonBlock and the other processes will be placed inside the GameBlock.

Associations between classes are mapped to channels and signal routes. The association between the two classes GameBlock and DemonBlock will become a channel between the corresponding blocks. The association between the classes Main and the ProcessDemon will result in a path of communication between the resulting processes.

The inheritance relationship between the Game class and its sub-classes will become inheritance relationships between the corresponding processes as well. Since associations are inherited in UML but signal routes or channels are not inherited in SDL, the UML2SDL utility creates signal routes/channels for inherited associations. This results in signal routes between the process Main and Game, as well as between Main and the sub-processes of Game.

Classes with the stereotype «actor» is mapped as communication with the environment. In our example the class Player will become a channel to the environment. The operations of the class Player will be signals to the environment.

The State Charts

There exist two state charts named Main and Game in the module that will be converted, see Figure 357. The intention of using state charts in the analysis is to get an overview of the behavior of important classes, see Figure 358. The state charts will be transformed together with the class diagram. Since two «process» classes Main and Game exist the result of the UML to SDL conversion will be two complete process descriptions.

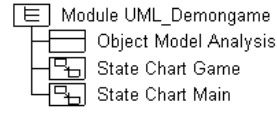


Figure 357: The module that will be converted.

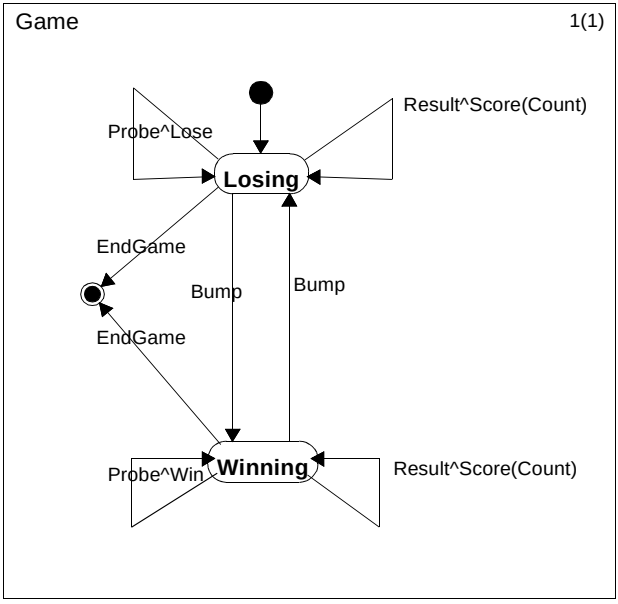


Figure 358: The behavior of Game.

The Design Model

The nature of software engineering always requires design information to be added to the analysis at some stage, because we can never solve a problem without providing a solution. This is also true when the UML2SDL utility is used.

Typically, information that needs to be added is behavior that is not described at the UML level. For instance, there is no good way of describing behavior that is redefined in a sub-class at the UML level. For example, such behavior is added to the sub-processes of process Game. Also, there was no need of modeling the simple behavior of ProcessDemon at the stage of analysis, but we have to do that in the design.

Besides adding the basic behavior, we also need to provide a full design. This includes actions like introducing variables, timers, etc. For example, in the process Main we need to describe how the process is responsible for the creation of Game processes, see Figure 359.

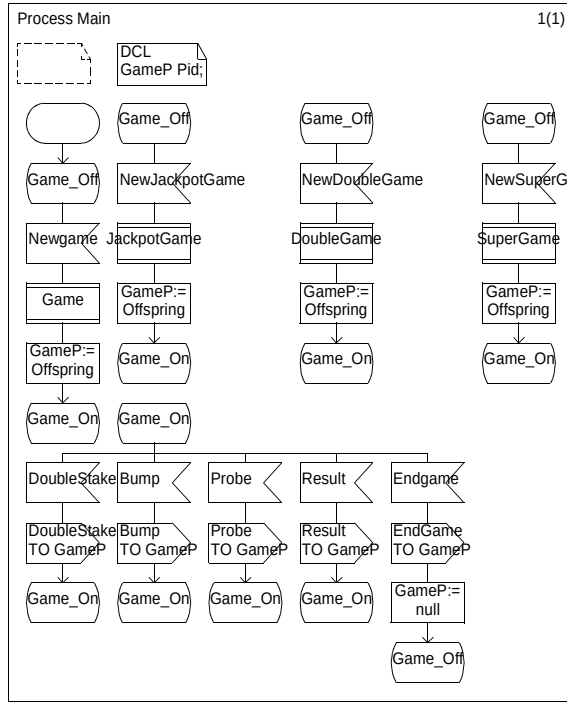


Figure 359: The complete process Main

When these steps are taken, the Demon game should be complete and possible to simulate like other SDL systems.

Summary

The intention with the UML2SDL utility is to automate the transfer from the analysis model to the design model as much as possible. The transfer will most likely include manual steps, but in theory it is possible to generate an SDL model which is detailed enough to simulate.

Chapter 42 The UML2SDL Utility

Chapter

43 *Using the SDL Editor*

This chapter describes the functionality, menus, dialogs and windows of the SDL Editor. You can also find information about how to use the editor for creating and drawing SDL (Specification and Description Language) diagrams.

General

Editor Information

The editor described in this chapter handles SDL diagrams.

Another editor is capable of handling four different types of diagrams, namely the Object Model (OM) diagrams, State Chart (SC) diagrams, Message Sequence Chart (MSC) diagrams, and High-level MSC (HMSC). That editor is described in chapter 39, *Using Diagram Editors*.

SDL Diagrams

The SDL Editor can handle any number of SDL diagrams of any type. Virtually all of the Z.100 recommendation is supported.

The SDL Editor supports three kinds of SDL diagrams:

- Interaction Diagrams
- Flow Diagrams
- Overview Diagrams

The editing functionality that is available depends on what type of diagram the SDL Editor is handling.

This chapter describes the functionality the SDL Editor provides when you edit interaction and flow diagrams.

In Figure 360–Figure 362, you can find an example of interaction diagrams (in the example there is one system diagram and two block diagrams) and the resulting overview diagram.

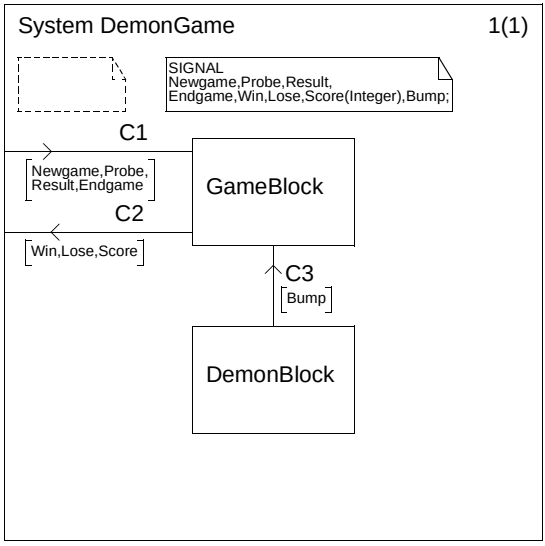


Figure 360: System diagram

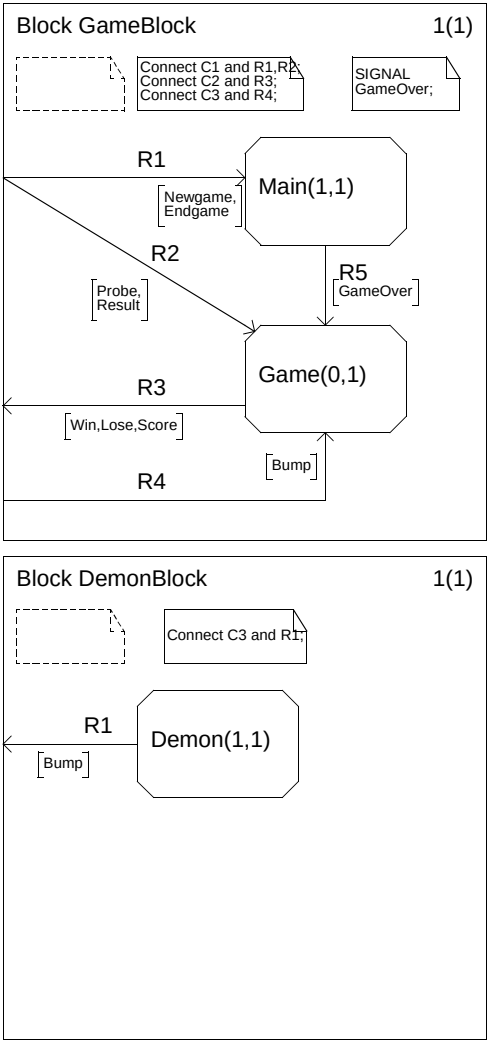


Figure 361: Block diagrams

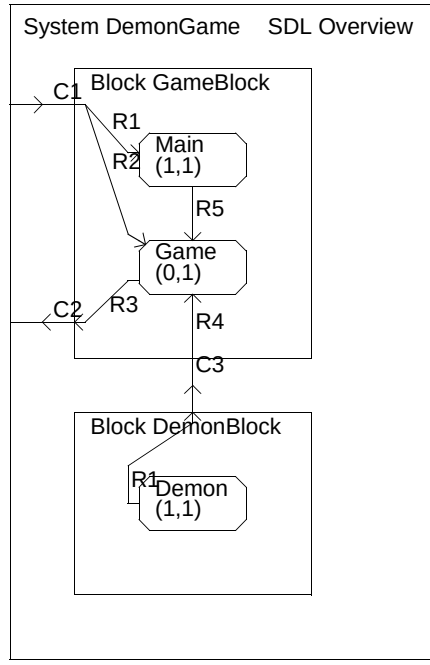


Figure 362: Resulting overview diagram

Interaction and Flow Diagrams

The SDL Editor works with SDL diagrams in graphical form, SDL/GR, and allows graphical editing of *symbols*, *lines* and *text*.

You may open multiple windows in an SDL Editor to be shown at the same time; each window will hold an SDL page. This can be useful if you want to edit different diagrams and pages and refer between them, or if you want to see the interface between two processes on different hierarchical levels. This is the normal mode in which diagrams are edited.

Interaction diagrams are: *system, system type, block, block type, substructure, package* diagrams.

Flow diagrams are: *process, process type, service, service type, procedure, operator, macro* diagrams.

Overview Diagrams

You can also use the SDL Editor to view *SDL overview diagrams*. An SDL overview diagram consists of a number of diagrams that are built up into a diagram subtree in which the symbols are drawn in a nested fashion, giving a more comprehensive overview of a required set of diagrams. You may have multiple SDL overview pages associated to one SDL system, at different SDL levels, and thus presenting different views of the system. You generate overview diagrams in the Organizer (using the *SDL Overview* command), and you can open the overview diagrams for display in the SDL Editor. Various parameters allow you to generate overview diagrams to meet your requirements.

Once an SDL overview diagram is generated, you cannot modify it in the same way that you would normally edit interaction and flow diagrams (unless you regenerate it). However, you can do limited editing on the text of overview diagrams in the SDL Editor.

Essentially, the editor works in read-only mode with regard to graphical operations. General restrictions in editing Overview diagrams are:

- There is no *symbol menu.*
- Only *textual elements* can be modified, not symbols and lines.

SDL Pages

The pages that the SDL Editor displays are always contained within an SDL diagram. An SDL diagram can contain any number of pages, but, must contain at **least one page**, and can be any of the following types.

- *Block interaction, process interaction, service interaction, package* (the *interaction pages*)
- *Graph, service, procedure, macro, operator* (the *flow pages*)
- *Overview*

Relationship between Diagrams and Pages

Pages and diagrams must be associated with each other according to the rules of SDL. There follows a list of what type of pages can be added to different diagrams and what type of pages can be pasted into those diagrams.

Diagram type	Page types that can be added to the diagram	Page types that can be pasted to the diagram
system	block interaction	block interaction package
block	block interaction process interaction	block interaction process interaction package
substructure	block interaction	block interaction package
service	service	service graph procedure macro operator
process	graph service interaction	graph service interaction service procedure macro operator
procedure	procedure	procedure service graph macro operator
system type	block interaction	block interaction package
block type	block interaction process interaction	block interaction process interaction package
service type	service	graph procedure macro operator

Diagram type	Page types that can be added to the diagram	Page types that can be pasted to the diagram
process type	graph service interaction	graph service interaction service procedure macro operator
macro	macro	macro service graph procedure operator
operator	operator	operator service graph procedure macro
package	package	package block interaction
overview	overview (one page only)	-

SDL Page Order

The SDL pages that are contained in an SDL diagram are listed and handled according to the order they were added when they were created.

This order is reflected in some of the menu choices that are related to SDL pages. Also, the structure displayed by the Organizer will adopt the same order. See "Chapters" on page 47 in chapter 2, *The Organizer*.

SDL pages can be renamed and rearranged by the *Edit* menu choice in the *Pages* menu.

Tracing Simulations (Graphical Trace)

Graphical trace, GR trace, is a method to follow the execution of transitions in SDL/GR source diagrams. It is mainly intended to be utilized together with the simulator's *Step-Symbol* command and should normally only be used for a small number of processes to limit the amount of information displayed. The GR trace facility will always show the next SDL symbol within the transition to be executed. After a nextstate or stop operation (e.g. between two transitions), the nextstate or stop symbol is still selected.

The GR tracing is activated from the SDL Simulator. The SDL Editor selects the symbol currently being executed. The Simulator automatically highlights the symbol and displays it. You can use either SDL to see what is being traced, or MSC to see the interaction between processes.

If you use an MSC to trace the simulation, the instances concept of MSC is mapped to the instances concept of SDL processes. The mapping rules which govern how SDL events are transformed into MSC symbols, lines and textual elements are described in "Mapping Between SDL and MSC" on page 1726 in chapter 39, *Using Diagram Editors*.

GR trace will take place in an SDL Editor window containing the appropriate diagram. If the SDL Editor does not hold the current page, but if the diagram can be found in the Organizer, then the page will be opened by the SDL Editor.

If the Organizer cannot find the diagram, or the SDL Editor cannot find an appropriate symbol to select, due to, for example, modifying the diagrams after the generation of the simulator, error messages will be issued by the Organizer or SDL Editor, but the simulation program will continue to execute without tracing graphically.

Setting Breakpoints in Simulations

During simulations, the SDL Editor can show all breakpoints that have been set in the Simulator. It is also possible to set and remove breakpoints directly in the SDL Editor by using the special *Breakpoints* menu that will be shown while this function is operating. The functionality is activated by the *Connect-To-Editor* command in the Simulator.

Compliance with ITU Z.100

The SDL Editor complies with the SDL rules as defined in the ITU Z.100 recommendation. Virtually all parts of the recommendation are supported by the SDL Editor. More information on the SDL support compared to Z.100, see "Compatibility with ITU SDL" on page 2 in chapter 1, *Compatibility Notes, in the Release Guide*.

Syntax Rules when Editing

Some of the SDL syntax rules are enforced during editing when you create diagrams, add pages to these diagrams, add symbols and lines to your pages and edit the text inside the symbols and lines.

Next follow the syntax checks that are performed by the SDL Editor:

- The SDL Editor checks that names on diagrams and pages adhere to the SDL definition.
- The type of a page that you add to a diagram must comply with the diagram type.
- The set of the symbols you are allowed to add to a page is in accordance to the page type. Also, the SDL Editor ensures that symbols are interconnected in a way that is syntactically correct.
- The text that is entered into a symbol or a line is instantly checked for syntax errors. When an error is located a red bar will underline the text where the error occurs.

Turning Syntax Checking On and Off

The syntax checking for entered texts, for what symbols you are allowed to use, and how you can interconnect symbols can be disabled and enabled again.

To disable or enable syntax checking:

1. The operation applies on one diagram at the time. Therefore, make sure you are editing the correct diagram.
2. Select the *Diagram Options* menu choice from the *View* menu. In the dialog which is issued, toggle the *Layout Syntax Check* and/or *Textual Syntax Check* button on or off and click *OK*.

Note: No Retroactive Syntax Checking

Turning the switch from off to on will not perform a retroactive syntax check for not allowed interconnection of symbols. Only the objects that you insert while syntax checking is enabled are checked. However, the syntax check on texts will be performed on all the texts when syntax check is set to on.

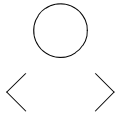
Syntax checking in Kernel Heading and Reference symbol

The syntax check for texts in the kernel heading or any reference symbol is always on independent of the diagram options setting.

Pasting and Syntax Checking

Only symbols that are valid according to what is displayed within the symbol menu may be pasted into a page. Non-valid symbols will be omitted whether or not syntax checking is enabled.

Identical Symbols – Different Syntax



SDL contains some symbols which appear identical, but are distinguished by their syntax. The symbols are:

- The **in connector** and the **out connector**
- The **continuous signal** and the **enabling condition**.

When layout syntax checking is on, the SDL Editor automatically determines from the context what kind of symbol it is. If syntax checking is off, the symbols are always treated as in connector and enabling condition, respectively.

Note: No Layout Check means Syntax errors

When disabling the layout syntax check and using connectors in the diagram mean that it will not be possible to analyze the diagram without errors as all connectors are treated as in connectors. Thus never turn off this switch if you want to analyze the diagram.

SDL Grammar Help

The SDL Editor provides a versatile context-sensitive support function – the *Grammar Help* window. The grammar help windo*w* is available to assist you when entering and editing the syntax of SDL text elements

that are correct according to Z.100 definition. It simplifies writing statements with the correct SDL syntax.

The SDL Grammar Help facility is described in "Using Grammar Help" on page 1972.

Signal Dictionary Support

The SDL Editor also provides a means of access to a *signal dictionary* where the signals that are already defined can be accessed. The Signal Dictionary window provides functions for listing SDL signals that are available when looking up or down in the SDL hierarchy or by looking at the current diagram. Also, you can ask the tool to list signals that are used in the current diagram or defined and visible according to the SDL scope rules.

Furthermore, the signal dictionary has the ability to import messages from a Message Sequence Chart (each MSC message will be mapped to an SDL signal) and to import an external signal dictionary.

For more information on this topic, see "Using the Signal Dictionary" on page 1983.

The SDL Editor User Interface and Basic Operations

The SDL Editor User Interface

The SDL Editor consists of a main window and three auxiliary windows:

- The *Grammar Help window* where SDL textual grammar support helps you entering correct textual expressions.
- The *Signal Dictionary window* which gives you access to signals already defined in the SDL hierarchy.
- The *Entity Dictionary Window* which provides you a name reuse facility as well as information for link endpoints.

The SDL Editor also contains the *symbol menu*, where you select the symbols (objects) that are to be inserted into the page, and the *text window*, where you may edit textual objects.

The general user interface is described in chapter 1, *User Interface and Basic Operations*.

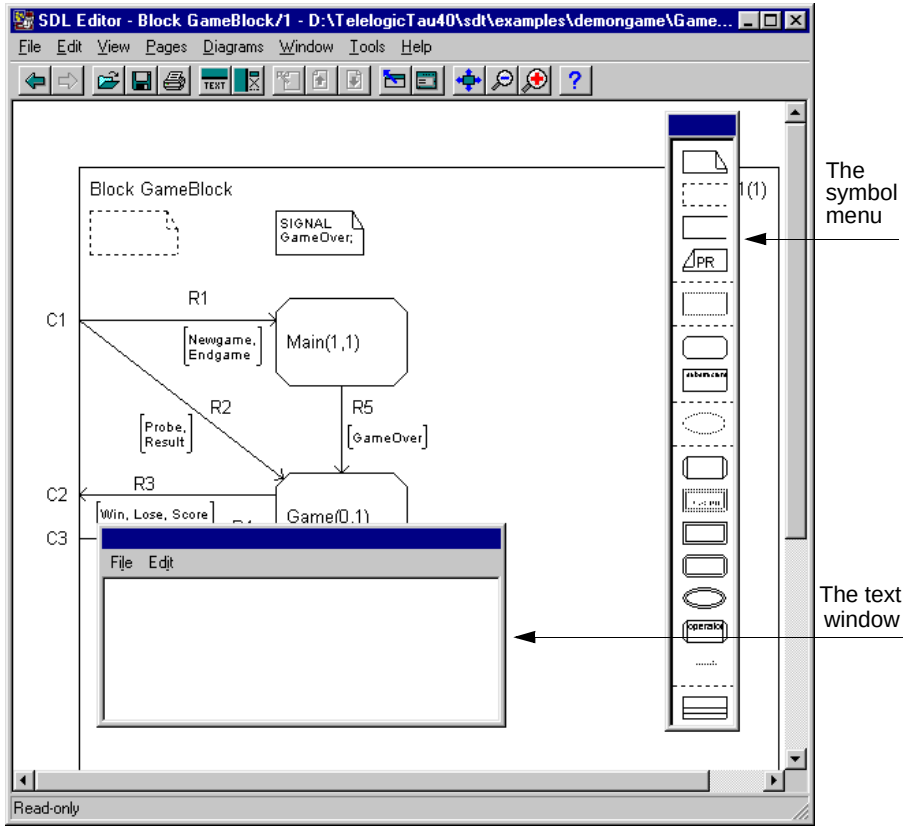


Figure 363: The SDL Editor window **(in Windows)**

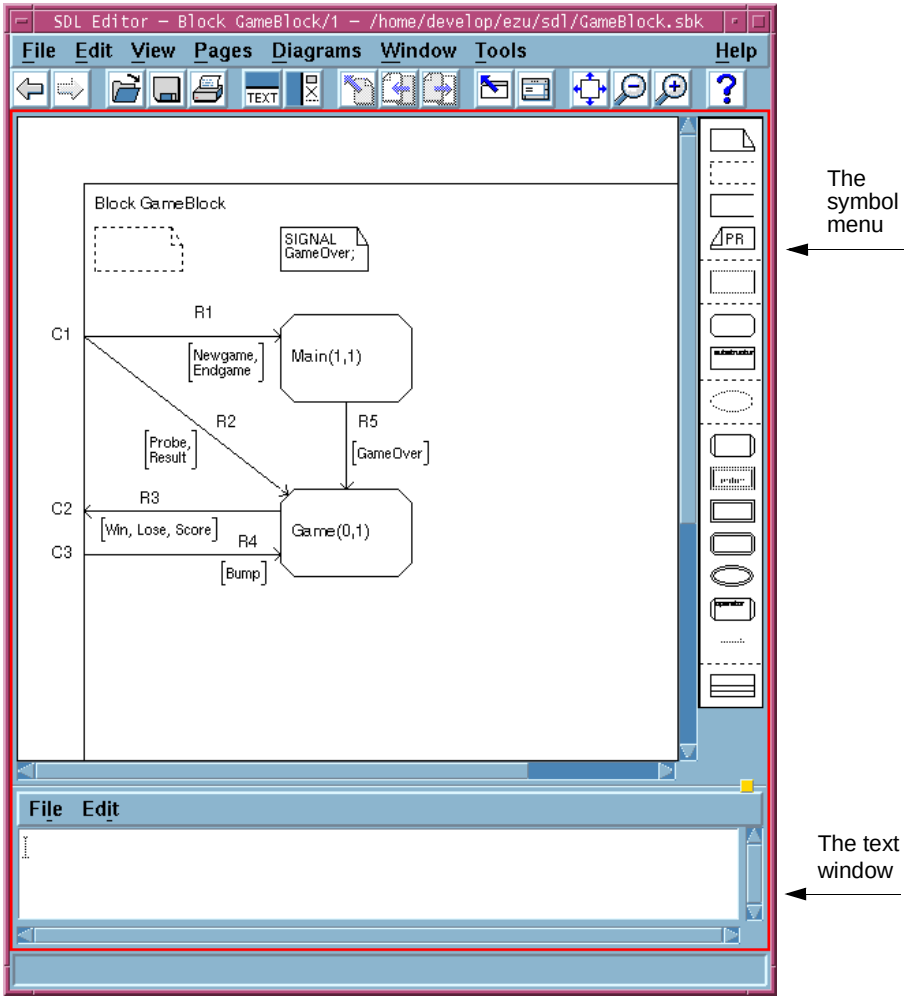


Figure 364: The SDL Editor window **(on UNIX)**

Drawing Area

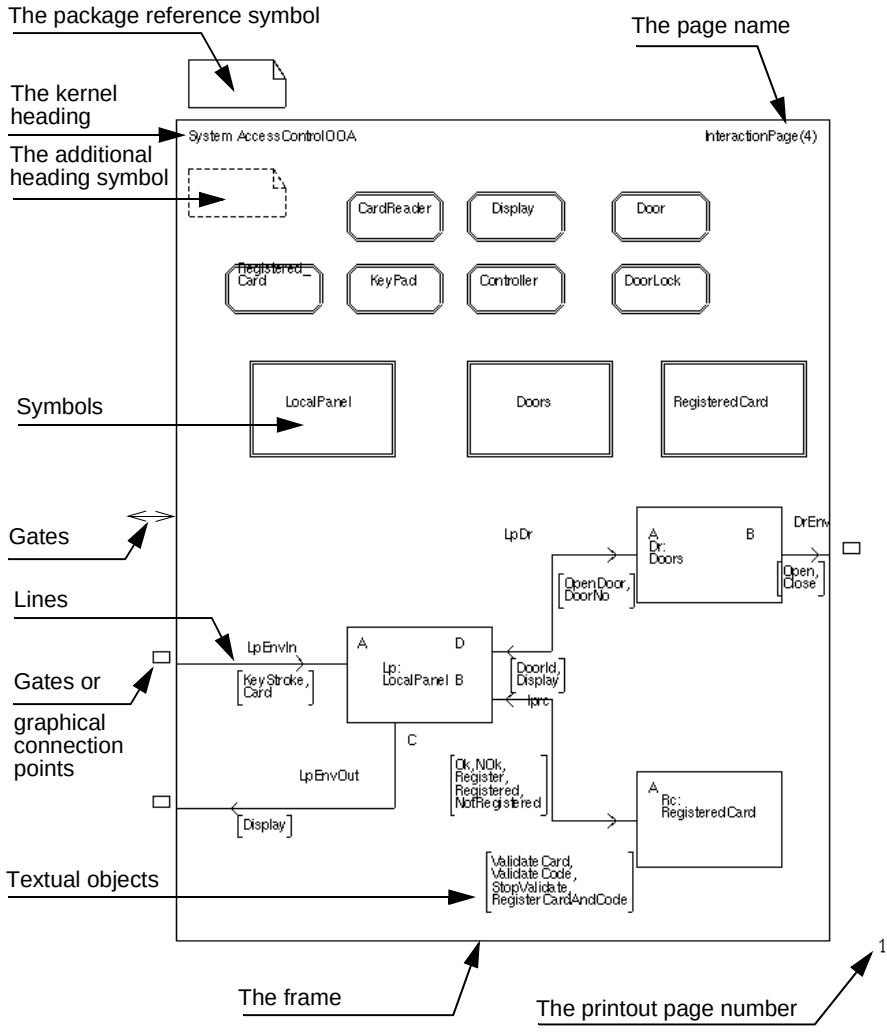


Figure 365: The Drawing Area

The *drawing area* (see Figure 365) is the part of the window that displays the symbols, lines and text that constitute an SDL page (an SDL diagram can contain multiple pages). Most of the graphical notation that is used in the SDL Editor is inherited from the Z.100 definition. In addition, some tool specific notations have been added to facilitate the work with the SDL Suite tool set in general, and the SDL Editor in particular.

The Drawing Area Boundaries

The drawing area is delimited by its boundaries, which correspond to the size of the SDL page. Within a diagram, each page has an individual size. No objects are allowed to be placed outside these boundaries. The drawing area uses a light background color, while the area outside the drawing area uses a grey pattern.

The Frame

You can select and resize the frame. All objects, except the package reference symbol, gates and printout page number must reside entirely within the frame.

The Kernel Heading

The kernel heading identifies the diagram type and its name. Alternatively, an SDL qualifier expression can be used. You can edit but not move the kernel heading. The SDL Editor performs a textual syntax check based on the grammar used in this heading.

The SDL Suite supports the following syntaxes for the kernel heading:

Case	Syntax
1.	`SYSTEM <name> [: <type expression>]`
2.	`BLOCK <identifier>`
3.	`PROCESS <identifier> [<number of instances>]`
4.	`SERVICE <identifier>`
5.	`<procedure preamble> PROCEDURE <identifier>`
6.	`SUBSTRUCTURE <identifier>`
7.	`MACRODEFINITION <name>`

Case	Syntax
8.	SYSTEM TYPE <identifier>
9.	[<virtuality>] BLOCK TYPE
10.	[<virtuality>] PROCESS TYPE
11.	[<virtuality>] SERVICE TYPE
12.	OPERATOR <operator identifier>
13.	PACKAGE <name>

For an explanation and reference to the notation used in the table above, see the Z.100 recommendation.

The kernel heading is repeated trough all pages contained in an SDL diagram.

The Page Name

The page name identifies the name of the SDL page and, within parentheses, the total number of SDL pages contained in the current SDL diagram.

The contents of the page name are assigned by the tool. You can not select, move, or edit the page name.

Printout Page Number

This object is created by the SDL Editor to inform you about the physical page numbering that will be the result of an SDL page which is larger than the paper format that is defined. You can neither select nor edit it. Page numbering follows a “down first, then right” fashion.

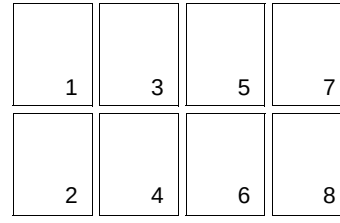


Figure 366: The page numbering in the SDL Editor

Grids

The SDL Editor uses two grids for an easy positioning of symbols, lines and textual elements.

- The symbol grid, that can be set by an SDL Editor preference, has a minimum resolution of 5 * 5 mm. All symbols adhere to the symbol grid.
- The line and text grid has a resolution of 2.5 * 2.5 mm, i.e. half the symbol grid. All lines and textual objects adhere to the line grid.

Default Size of Symbols

The symbols are assigned default sizes when added to the drawing area:

- For interaction pages, the predefined size of a symbol is by default 30 * 20 mm.
- For flow pages, the predefined size of a symbol is by default 20 * 10 mm. The width / size relationship is always 2:1.

You can change the default symbol size in the SDL Editor preferences.

> **Note:**
>
> The size of a class symbol is automatically calculated by the SDL Editor, and may thus differ from the default size.

Color on Symbols

For each symbol type there is a preference for setting the color of the symbol. It is only the graphical part of the symbol and not the associated text(s) that will use the color setting. **On UNIX**, the setting is only valid on screen and all symbols will use the black color when printed on paper. **In Windows**, when using MSW Print the colors will be sent to the printer as well. For more information, see *"The SDL Suite Specific Preferences" on page 267 in chapter 3, The Preference Manager*.

Keyboard Accelerators

In addition to the standard keyboard accelerators, described in "Keyboard Accelerators" on page 35 in chapter 1, *User Interface and Basic Operations*, the SDL Editor features the following:

Accelerator	Reference to corresponding command
`Ctrl+D`	Opens the next page in the diagram, similar to "<Page Name>" on page 2020
`Ctrl+G`	"Split Text" on page 2009
`Ctrl+I`	"Insert Paste" on page 2007
`Ctrl+L`	"Show High-Level View/Show Detailed View" on page 2015
`Ctrl+M`	"Mark as Important/Mark as Detail" on page 2011
`Ctrl+U`	Opens the previous page in the diagram, similar to "<Page Name>" on page 2020
`<Delete>`	"Clear" on page 2008 (i.e. remove, delete)
`Ctrl+1`	"Show Organizer" on page 15 in chapter 1, *User Interface and Basic Operations*
`Ctrl+2`	"Connect to Text Editor" on page 2025

Quick-Buttons

In addition to the generic quick-buttons described in "General Quick-Buttons" on page 24 in chapter 1, *User Interface and Basic Operations*, the SDL Editor tool bar contains the following quick-buttons:

Text window on / off
Toggle the text window between visible and hidden, as described in "Window Options" on page 2016.

Symbol menu on / off
Toggle the symbol menu between visible and hidden, as described in "Window Options" on page 2016.

Reference page
Open the page where this diagram is referenced, similar to "Edit Reference Page" on page 2020.

Log Window
Pop up the Organizer Log window.

Previous page
Open the previous page in the diagram, similar to "<Page Name>" on page 2020.

Next page
Open the next page in the diagram, similar to "<Page Name>" on page 2020.

Toggle scale
Toggle the scale between a scale to show the complete page in the window and a scale of 100%.

Scrolling and Scaling

You can scroll the view vertically and horizontally by using the scrollbars. The view may also be scrolled automatically when you move the cursor beyond the current view, for example when you move an object or add a symbol.

If you move the cursor close to the edge of the current view, the automatic scrolling is slow. If you move it further beyond, the scrolling is quicker.

You can scale the view by specifying a scale or by zooming in and out.

To specify a scale:

1. Select *Set Scale* from the *View* menu.
2. In the dialog that will be opened, you can either:
   - Use the slider for setting the scale and then click the *Scale* button.

   - Click the *Overview* button to adjust the drawing area to the size of the window. (This has the same effect as the *Scale Overview* quick-button.) The smallest scale is 20%.

To zoom in or out:

- Click the quick-button for *zoom in* or *zoom out*.

Moving Selection with Arrow Keys

You can move the selection using arrow keys in the SDL editor if there is one single symbol or line selected. (In this context, we regard a flow symbol together with its attached flow lines to be one symbol.)

Note that you can only move the selection when in symbol editing mode. In text editing mode, the arrow keys will move the cursor position within the text. To get into symbol editing mode, click on the symbol outside of a text area. If you are in text editing mode, you can go to symbol editing mode by pressing the escape key. If you are in symbol editing mode, you can go to text editing mode by pressing the tab key. Pressing the tab key immediately after that will either select another text for the same symbol to edit or go back to symbol editing mode.

You can move to the next page by moving forward from the last symbol on the current page. You can move to the previous page by moving backward from the first symbol on the current page.

Two shortcuts are available for fast-forward moving to colored symbols (symbols not using black as a border color and white as a background color) on pages in the current diagram:

- Shift+arrow key to move to the next/previous colored symbol.
- Shift+alt+arrow key to move to the next/previous colored symbol on another page than the current page.

Symbol Menu

The *symbol menu* contains the SDL symbols that you can place into the drawing area.

On UNIX, the *symbol menu* is a fixed-sized, non-moveable auxiliary window, associated with the drawing area and placed to the right of it. Each editor window has its own symbol menu.

In Windows, the symbol menu is a fixed-sized, moveable window that can be placed anywhere on the screen, not necessary within the limits

of the editor window. A single symbol menu is shared by all instances of the editor currently running.

The symbol menu can be made invisible and visible again with a menu choice, *Window Options*, or a quick-button **In Windows:** When visible, the symbol menu will always be placed on top of the editor window.

Basically, when you select a symbol in the symbol menu and click it into the drawing area it is added to the diagram. How to work with symbols is described in "Working with Symbols" on page 1901.

The contents of the symbol menu depends upon the type of diagram that is displayed in the SDL Editor window and the settings of the SDL Editor preferences depends upon whether the symbols should be used or not.

- For *process*, *process type*, *procedure*, *macro definition* and *operator diagrams* (the *flow* diagrams), the default symbol menu looks like the left picture in Figure 367.
- For interaction diagrams (*system*, *block*, *substructure*, *service*, *package*, *system type*, *block type* and *service type*), the default symbol menu looks like the right picture in Figure 367.

For a reference to these symbols, see "Symbols on Interaction Pages" on page 1885 and "Symbols on Flow Pages" on page 1887.

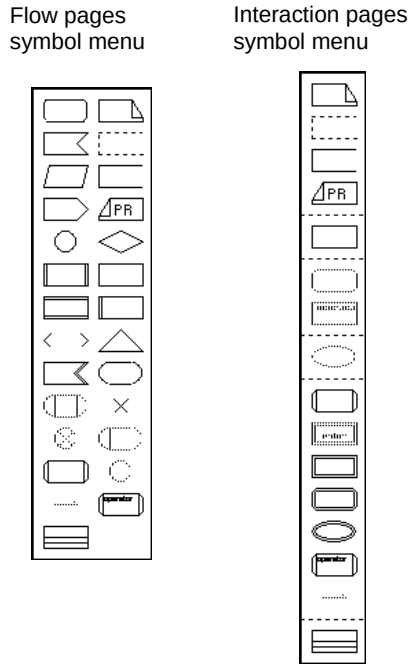


Figure 367: The Symbol menu

Syntax Checking on SDL Symbols

If layout syntax checking is enabled, some of the symbols displayed in the *symbol menu* may be dimmed. This indicates that those symbols are not valid within the current context, and that you cannot select them.

- When layout syntax checking is disabled, you can select all symbols.

- When layout syntax checking is enabled, the following rules apply:

Type of page	Unusable symbols (dimmed)
System / System type	process reference service reference block substructure reference system type reference gate
Block diagram / Block interaction page	process reference service reference block substructure reference system type reference gate
Block diagram / Process interaction page	block reference service reference system type reference gate
Service	procedure start procedure return macro inlet macro outlet gate
Process diagram / Graph page	procedure start procedure return macro inlet macro outlet gate
Process diagram / Service interaction page	block reference process reference block substructure reference system type reference block type reference process type reference gate

Type of page	Unusable symbols (dimmed)
Procedure	start stop macro inlet macro outlet gate
Block type diagram / Block interaction page	process reference service reference block substructure reference system type reference
Block type diagram / Process interaction page	block reference service reference system type reference
Macro	gate
Service type	procedure start procedure return macro inlet macro outlet
Process type diagram / Graph page	procedure start procedure return macro inlet macro outlet
Process type diagram / Service interaction page	block reference process reference block substructure reference system type reference block type reference process type reference

Type of page	Unusable symbols (dimmed)
Operator	state input save output procedure call enabling condition / continuous signal create request priority input procedure reference start stop macro inlet macro outlet gate
Package	block reference process reference block substructure reference service reference gate

Working with Diagrams

This section describes the methods you use when performing the following tasks on SDL diagrams:

Creating a Diagram

You can create diagrams from within other diagrams. When created, they form part of the current active system in the SDL Editor, and their presence is instantly reflected in the Organizer. This is achieved by inserting SDL diagram *reference symbols*.

You can also create diagrams as separate unrelated new entities. They will however not be incorporated in the Organizer.

Creating a Related Diagram from Another Diagram

1. Open the SDL diagram from where to start.
2. Insert the SDL reference symbol.
3. Assign the symbol a name.
4. Deselect the reference symbol. The Organizer's diagram structure is updated accordingly.
5. Double-click the newly added reference symbol. You should specify what action to perform, either:
   - Create a new diagram in the SDL Editor from scratch.
   - Copy an existing file.

   Select an option and click *OK*.

The SDL Editor will now create a new diagram, possibly a new window, and display the newly created diagram.

6. Save both the parent and the child diagram later on in your editing session, to make changes permanent.

Example in an Organizer Structure

As an example, in the Organizer structure in Figure 368, a system is shown with only one block diagram.

Figure 368: A basic system shown in the Organizer

Suppose the system diagram is being edited. We now add a new block reference symbol, name it Block_B and deselect it. The Organizer diagram structure becomes:

Figure 369: Having added a reference symbol

The SDL Suite shows, with the `[unconnected]` identifier, that the newly added reference symbol is not yet connected to any physical file.

Now, double-click the Block_B symbol. Specify whether you want to create a new file or to connect the newly created block to an existing file.

Make sure the *Show in Editor* radio button is on and click *OK*. In some cases a new dialog appears, where you are to specify the name and type of the first SDL page to add to the diagram (an SDL diagram must contain at least one SDL page).

Assign a page name (see "Adding a Page" on page 1954) and click *OK*.

A new block diagram with the name Block_B is created. The SDL Editor responds by issuing a window showing this diagram. Also, the Organizer's diagram structure is updated, showing the new diagram icon, inserted at the corresponding place in the SDL hierarchy.

Figure 370 on page 1871 shows the result:

- A newly created diagram, empty except for the diagram heading
- The resulting diagram structure
  - The parent diagram needs to be saved (marked with a gray pattern)
  - The child diagram needs to be saved.

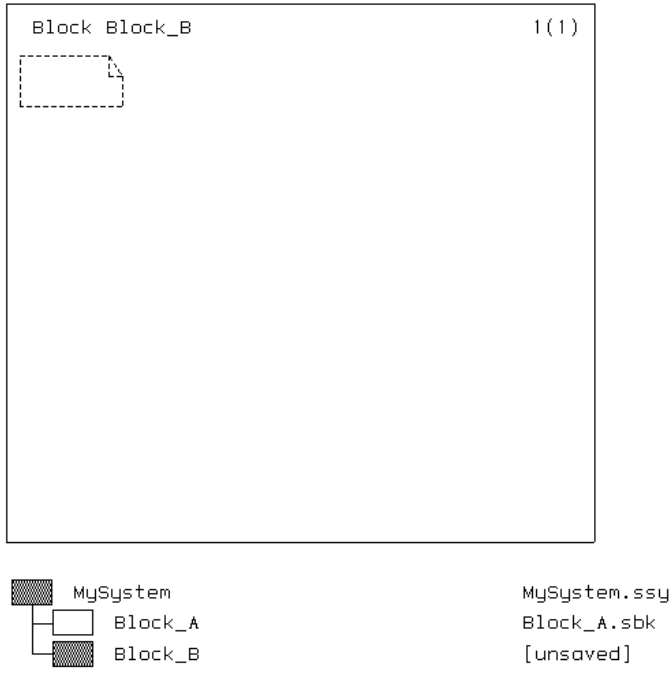


Figure 370: The newly created diagram and the resulting diagram structure

Creating an Unrelated New Diagram

1. Select *New* from the *File* menu of the SDL Editor. A dialog will be opened.
2. Enter the name of the new diagram in the *Diagram Name* field. The name should be in keeping with SDL naming conventions.

3. Choose the type of diagram from the *diagram type* menu.
4. Click *New*. In some cases a new dialog box is shown where you can add a page, as each diagram must have at least one page. See the section "Adding a Page" on page 1954 for this.

   A window on the newly created diagram is opened.

Including a Diagram

Including a Diagram into the SDL Hierarchy

When you have created a diagram, it needs to be brought into the SDL hierarchy to be available for future selection in the Organizer. Otherwise, you will only be able to open it from the SDL Editor.

1. Create the diagram using the *New* command from the SDL Editor. *Save* the diagram on file.
2. Open the diagram that will have a reference to the newly created diagram.
3. In the diagram you just opened, add a diagram reference symbol with a type and name that matches the newly created diagram. A new icon appears in the Organizer structure.
4. With the Organizer *Connect* command, connect the new icon with the file on which the diagram is stored. See "Connect" on page 92 in chapter 2, *The Organizer*.

Once a reference symbol is placed in the newly created page of the new diagram in the SDL Editor, the Organizer is updated to reflect this. This indicates that the SDL hierarchy is updated.

Including a Diagram into the Organizer

If a diagram is not managed by the Organizer (i.e. not contained in the Organizer structure) you must first include it into the Organizer, before you can start the SDL Editor.

1. Select the place in the Organizer where you want to add the SDL diagram. It will be added as a new root diagram after a selected document or chapter, or at the top level of a selected module.
2. Select the Organizer's *Add Existing* command from the *Edit* menu. A file selection dialog will be opened.
3. Select a SDL diagram file.

   The SDL diagram is added to the Organizer structure, and opened in the SDL Editor.

Opening a Diagram

There are several ways you can open a diagram for editing, depending on where you are and what you need to access.

Lock Files and Read-Only Mode

When you open a diagram or document file, i.e. *a.ssy*, a lock file *a.ssy.lck* is created. If another user tries to open the same diagram or document file before you close it, a dialog will appear informing the other user that the file is in use.

There are two choices in the dialog:

- Open the file in read-only mode.
- Reset the lock and open the file in read-write mode. (This alternative should only be used if the existing lock file is obsolete.)

The read-write mode is the normal editing mode. In read-only mode, you are not allowed to make any changes to the diagram.

You will also enter read-only mode if you open a diagram file that you do not have write permission for. The read-only mode is indicated by the words *read-only* in the window title.

If you want to edit a diagram that is in read-only mode, here are two alternative actions:

- Change the file access rights in the file system to give you write permission for the diagram file. Then change from read-only mode to read-write mode with the *Revert Diagram* operation.
- Use *File >Save As* to save the diagram in a new file with write permission.

Opening a Diagram File from the SDL Editor

1. Select *Open* from the *File* menu or click the quick-button for *Open*. A file selection dialog is opened.
2. Select a file in the files list and click *OK*. If the file is already opened by the SDL Editor, a message is issued.

> **Note:**
> The *status bar* at the bottom the SDL Editor provides information about the diagram type and name stored on the file which is selected in the file list.

Opening a Diagram from the Organizer

You can open a diagram in different ways:

- Select an SDL diagram icon or the text line next to it in the Organizer. Select *Edit* from the Organizer's *Edit* menu.
- Double-click on an SDL diagram icon in the Organizer.

Opening a Diagram – via Organizer – to Show Analyzer Reports

When analysis has been performed, you can show the source of Analyzer error reports in an SDL Editor.

- Select *Show Error* from the *View* menu in the *Organizer Log Window*. An SDL Editor window is opened with the source of error (i.e. an SDL symbol) selected.

Opening a Diagram via a Reference Symbol

You can open a diagram from the SDL Editor by double-clicking on a reference symbol, or by using the *Navigate* command.

1. Locate the SDL reference symbol referring to the diagram you want to open.
2. Double-click the symbol or select it and choose the *Navigate* command from the *Tools* menu. The diagram is opened.

Opening a Diagram from the Simulator

If you have performed a GR trace in the SDL Simulator, invoking the Show-Next-Symbol or Show-Previous-Symbol simulator commands will display either the next symbol to be executed or the previously executed symbol in a diagram in an SDL Editor window.

Opening a New (Nonexisting) SDL Diagram

To open a new SDL diagram, you either use an unconnected SDL diagram icon or add a new SDL diagram icon.

Opening an Unconnected SDL Diagram Icon

To open an existing but unconnected SDL diagram icon:

1. Select an unconnected SDL diagram icon in the Organizer.
2. Double-click the SDL diagram icon. A dialog is opened.
3. Make sure the *Show in editor* option is set. Click *OK*. A new, empty diagram is opened.
   - To copy the contents of an existing SDL diagram file before opening the new diagram, select the option *Copy existing file* and enter a filename.

Opening a New SDL Diagram Icon

To add a new SDL diagram icon to the Organizer and opening that icon:

1. Select the place in the Organizer where you want to add the SDL diagram. The diagram will be added as a new root diagram after a selected document or chapter, or at the top level of a selected module.
2. From the Organizer's *Tools* menu, select the *Editors > SDL Editor* command. A new SDL diagram icon is added to the Organizer structure, and the new diagram is opened in the SDL Editor.
   - Alternatively, select the Organizer's *Add New* command from the *Edit* menu. A dialog is opened. Select a SDL diagram type, enter a name in the *New document name* field and click *OK*. The *Add Page* dialog is displayed, see "Adding a Page" on page 1954.

Opening an Existing SDL Page

To open a specific page, the SDL pages must be visible in the Organizer diagram structure.

1. To display page icons in the Organizer, select the Organizer's *View Options* command from the *View* menu. In the dialog, make sure *Page symbols* is highlighted and click *Apply*.
2. Select the icon representing the SDL page in the Organizer.
3. Double-click the SDL page icon. The SDL diagram page is displayed.

Saving a Diagram

There are several ways to save a diagram. The effect is saving any changes made to either one specific diagram, or all diagrams modified during the current session.

Saving a Diagram in the SDL Editor

- Select *Save* from the *File* menu or click the quick-button for *Save.*

  If the diagram is newly created, a file selection dialog is issued where you can specify the file to save the diagram on.

Saving All Diagrams in the SDL Editor

You can make a global save of all the diagrams that are open in the current session and have been modified.

- Select *Save All* from the *File* menu. All diagrams which are modified are saved.

Saving a Copy of an SDL Diagram

You can save the diagram being edited into another file. You may either continue working with the original file or with the newly created copy.

Continue Working with the New Copy

1. Select *Save As* from the *File* menu. A file selection dialog is issued.
   - The file name which is suggested is the same file as the original was stored on, appended with an integer suffix (1,..., N) in order to build up a unique file name.
2. Click *OK*. The SDL diagram is copied to the specified file, the file containing the original is closed, and the window where you made the save remains open for editing the newly created file.

Continue Working with the Original

1. Select *Save a Copy As* from the *File* menu. A file selection dialog is issued.
   - The file name which is suggested is the same file as the original was stored on, appended with an integer suffix (1,...,N) that guarantees a unique file name.
2. Click *OK*. The SDL diagram is saved on the specified file, and the window where you made the save remains open for editing.

Closing a Diagram

Closing a diagram also means closing all instances of all windows displaying any page contained in that diagram.

- Select *Close Diagram* from the *File* menu.

  If you have modified the diagram, a *Close* dialog is shown where you can save the modifications before closing the diagram.

Printing a Diagram

You can print an entire SDL diagram or a selection of pages or objects contained in the diagram. You print multiple SDL diagrams from the Organizer.

See *"The Print Dialogs in the SDL Suite and in the Organizer" on page 316 in chapter 5, Printing Documents and Diagrams*.

Displaying an Opened Diagram

You can open the diagram that is currently read by the SDL Editor. (The diagram that you would like to see may reside in a window which is not on top of the screen.)

The *Diagrams* menu shows all SDL diagrams and pages (up to a maximum of the last nine to have been used). If more than nine pages are open, a tenth menu choice, *List All*, provides access to a list dialog where all diagrams and pages are listed.

Reorganizing a Diagram

The SDL Editor can reorganize the layout of an SDL diagram, for instance if you run out of space on any of the pages that build up the diagram.

To reorganize an SDL diagram:

1. Display the diagram you want to reorganize.
2. Select *Tidy Up* from the *Tools* menu. A dialog is opened.
3. Click *Tidy Up*. The SDL Editor reorganizes the diagram (this may take some time, depending on how complex your diagram is).
4. Check the diagram before continuing working with it (you can undo the operation from the *Edit* menu).

Transforming the Type of a Diagram

A useful operation when e.g. transforming to a more object-oriented version of the SDL system is to change the SDL88 diagrams into type diagrams and make instantiations of these types. The following shows some examples how to perform such an operation and similar ones

where you want to change the type of a diagram or want to work with an SDL structure that is almost a copy of an already existing hierarchy.

Example 1

Assume that the system S consists of block B, which contains process P. The exercise to solve is to transform the block into a block type BT and put this into a package P and instantiate the block B from BT.

There is no direct command to accomplish this task in the SDL Suite; instead there are many solutions, more or less complicated and time consuming. The proposed one is easy to use and will take advantage of the possibility to cut and paste complete sub-hierarchies, thereby eliminating loosing file connections and the need for making manual connections to existing files.

Solution: Open the system S, add a new block type symbol and name it BT. Open the block B and change the kernel heading from “block B” to “block type BT”. Note that the sub-hierarchy in the Organizer, previously below B, has now been moved to BT. In system S, change the text in the block symbol to “B:BT” to make it an instance of BT and cut the block type symbol to the clipboard. Create a package diagram named P and paste the block type into this diagram.

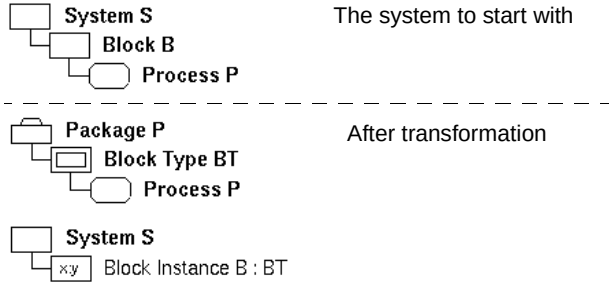


Figure 371: The Organizer view

To make the new system complete you need to add the text “use P;” in the package reference symbol in system S, define the gates inside the block type BT and use these gates to connect the existing channels for the block B. If needed, you can change the filename of the block type with the *Save As* command in the SDL Editor to use a filename that re-

sembles the block type name and also use the file name extension that conforms to the SDL Suite conventions.

Example 2

This is a variant of example 1. Assume we want to keep the original diagram but want to have a new block type BT in package P that is a copy of the existing B.

Solution: In package P place a new block type symbol and name it BT. Double click on it and in the appearing Edit dialog tick the *Copy existing file* option and use the file connected to the existing block diagram B. Click *Continue* in the warning about recommended file name extension. In the opened diagram, change the kernel heading to "block type BT".

For all diagram references in BT (in the example there is only one), use the *Connect* command in the Organizer to connect them to the existing files. In the Connect dialog be sure to use the *Expand Substructure* option to get automatic connections for possible sub-hierarchies.

As an alternative to connecting the diagram references you can copy the reference symbols from block B and paste them into block type BT, then you will get automatically all the connections for sub-hierarchies below the pasted references.

Note that in either case you will use the same file connection for process P in our example for both the original process and also for the process P referenced in the block type, meaning that if you change the contents of this file you will change both the behavior of the block B and the block type BT, which might lead to unexpected results. If you want unique file connections you should take a copy of the existing file, for example copy `P.spr` to `P1.spr`, and reconnect one of the processes to this new file instead.

Example 3

The SDL system consists of system S, block B, process P and procedure subPr inside P. You want to transform most of the process including its procedure subPr into a new procedure Pr defined in the system S.

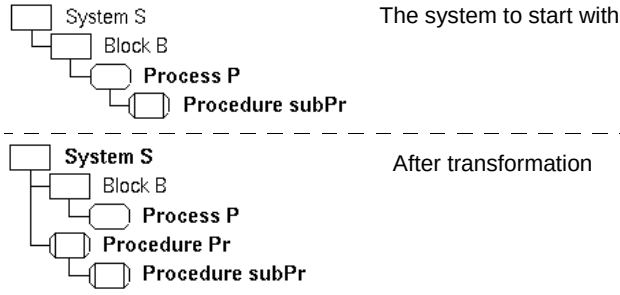


Figure 372: The Organizer view

Solution: This solution is similar to the solution to example 2. Open the system diagram S and place a procedure reference symbol named Pr. Double click it and open a copy of the existing file connected to process P. Edit the kernel heading to contain the text “procedure Pr”. Save the procedure as a new file. Connect all referenced procedures inside Pr or copy the procedure references from the process P to get all the sub-hierarchy file connections. Finally edit process P and procedure Pr to make them useful in their new context.

The symbol is dashed and the change is immediately reflected in the Diagram Structure (this is only true if the *Dashed Diagram* option is on in the Organizer’s *View Options*).

- You may at any time *undash* a dashed symbol. Select the symbol and select *Undash* from the *Edit* menu.
- If the symbol (prior to the dash operation) was connected in the Organizer the following dialog will appear:

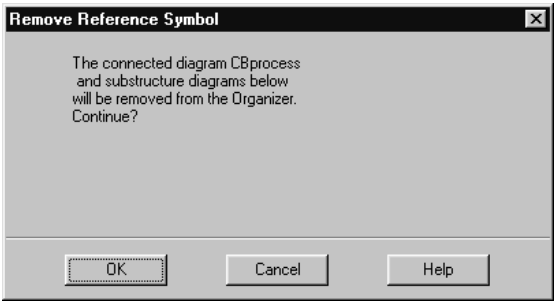


Figure 373: Removing substructure by dashing

- If you click *OK*, the substructure will be removed. However, you may undo this.

About Symbols and Lines

Following is a description of the components in the drawing area. See also appendix 44, *Symbols and Lines – Quick Reference, on page 2053*.

The Additional Heading Symbol

Symbol Appearance	Symbol Name	References to Z.100
	Additional heading	Z100: 2.2.5 Additional heading Z100: 6.1.1.3 Process type heading Z100: 2.4.4 Formal parameters, Signalset Z100: 6.3 Specialization Z100: 6.3.1 Inherits Z100: 6.3.2 Virtual, Redefined, Finalized Z100: 6.3.2 Virtuality constraint (atleast) Z100: 6.2 Actual context parameters Z100: 6.2 Formal context parameters Z100: 6.2.1 - 6.2.9 Formal context para

The additional heading symbol is not defined further according to Z.100. In the SDL Editor, it looks like a dashed text symbol or it is shown without any border; this is controlled by the Editor preference ScreenZ100Symbols. The symbol is editable and resizeable, but cannot be moved. Its intended use is to define:

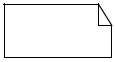
- Inheritance and specialization, using the following keywords:
  - `INHERITS`
  - `ADDING`
  - `ATLEAST`
- Formal parameters for a process / procedure, using the keyword `FPAR`

- Directives to the SDL to C Compiler. A directive uses the SDL comment notation and a hash character, such as:
  `/*#INCLUDE file.pr */`
- `SIGNALSET` and instance information.

The additional heading symbol is either repeated through all pages contained in an SDL diagram or it is only shown on the first page. This is controlled by the editor preference AdditionalHeadingOnlyOnFirstPage and can also be specified with the SDL Editor command *Diagram Options*.

Note that the combined text in the Kernel Heading and the Additional Heading symbol must use the syntax as specified for a heading in SDL/PR to be accepted in the SDL Analyzer.

The Package Reference Symbol

Symbol Appearance	Symbol Name	References to Z.100
	Package reference	`Z100: 2.4.1.2 Reference in Package` `Z100: 2.4.2 Reference in System`

The package reference symbol is available on system and package diagrams only. It is located outside the frame, on top and to its left. It contains references to package(s) containing definitions that are to be included in the SDL system.

You can select and resize, but not move the package reference symbol.

Other SDL Symbols

These are the symbols that describe the structure or behavior of the SDL diagram. They must be placed inside the frame.

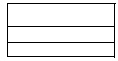
You can draw symbols in color, see "Color on Symbols" on page 1860.

Syntax Checking on Symbols

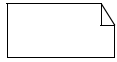
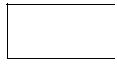
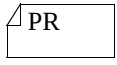
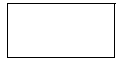
The SDL Editor checks that the symbols you add to a diagram are in accordance with the syntactic rules imposed by SDL. Symbols that are not allowed in a diagram / page of a specific type are dimmed in the symbol

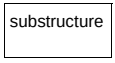
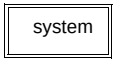
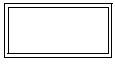
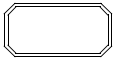
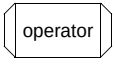
menu. Also the text entered is checked for syntax errors. See "Layout syntax check" on page 2017.

Symbols on both Interaction Pages and Flow Pages

Symbol Appearance	Symbol Name	References to Z.100
	Class	Not included (Z100 11/99)

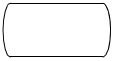
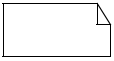
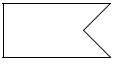
Symbols on Interaction Pages

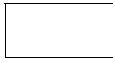
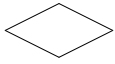
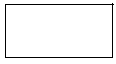
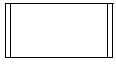
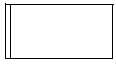
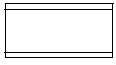
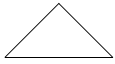
Symbol Appearance	Symbol Name	References to Z.100
	Text	Z100: 2.5.4 Signal Z100: 2.5.5 Signal list Z100: 5.2.1 Newtype Z100: 5.3.1.9 Syntype Z100: 5.3.1.13 Synonym Z100: 5.3.1.12.1 Generator Z100: 4.13 Remote variable Z100: 4.14 Remote procedure
	Comment	Z100: 2.2.6
	Text extension	Z100: 2.2.7 (depends on the symbol connected to)
	Text Reference	None
	Block reference	Z100: 2.4.2 Block definition Z100: 6.1.3.2 Block def based on block type Z100: 6.1.2 Type expression Z100: 6.2 Actual context parameters

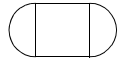
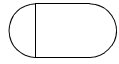
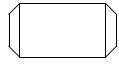
Symbol Appearance	Symbol Name	References to Z.100
	Process reference	Z100: 2.4.3 Process definition Z100: 2.4.4 Number of instances Z100: 6.1.3.3 Process def based on block type Z100: 6.1.2 Type expression Z100: 6.2 Actual context parameters
	Block substructure reference	Z100: 3.2.2
	Service reference	Z100: 2.4.4
	System type	Z100: 6.1.1.1
	Block type	Z100: 6.1.1.2
	Process type	Z100: 6.1.1.3
	Service type	Z100: 6.1.1.4
	Operator reference (an SDL/GR extension defined in the SDL Suite)	Z100: 5.3.2 Referenced operator in SDL/PR

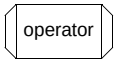
Symbol Appearance	Symbol Name	References to Z.100
	Gate	Z100: 6.1.4 Gate Z100: 2.5.5 Signal list

Symbols on Flow Pages

Symbol Appearance	Symbol Name	References to Z.100
	State or nextstate	Z100: 2.6.3 State Z100: 4.4 Asterisk state Z100: 4.5 Multiple appearence of state Z100: 2.6.8.2.1 Nextstate Z100: 4.9 Dash nextstate
	Text	Z100: 2.5.4 Signal Z100: 2.5.5 Signal list Z100: 5.2.1 Newtype Z100: 5.3.1.9 Syntype Z100: 5.3.1.13 Synonym Z100: 5.3.1.12.1 Generator Z100: 2.6.1.1 Variable Z100: 2.6.1.2 View Z100: 2.8 Timer Z100: 4.13 Remote variable Z100: 4.13 Imported variable Z100: 4.14 Remote procedure Z100: 4.14 Imported procedure
	Input	Z100: 2.6.4 Input Z100: 4.6 Asterisk input Z100: 4.14 Remote procedure input Z100: 6.3.3 Virtual transition z100: 5.4.3 Variable z100: 5.4.3.1 Indexed variable z100: 5.4.3.2 Field Variable
	Comment	Z100: 2.2.6

Symbol Appearance	Symbol Name	References to Z.100
	Save	Z100: 2.6.5 Save Z100: 4.7 Asterisk save Z100: 6.3.3 Virtual save
	Text extension	Z100: 2.2.7 (depends on the symbol connected to)
	Output	Z100: 2.7.4
	Decision	Z100: 2.7.5 Decision Z100: 5.3.1.9.1 Range condition Z100: 2.2.3 Informal text
	In-connector or out-connector	Z100: 2.6.7 In-connector Z100: 2.6.8.2.2 Out-connector
	Task, set, reset or export	Z100: 2.7.1 Task Z100: 5.4.3 Assignment Z100: 2.8 Set, Reset Z100: 4.13 Export
	Procedure call	Z100: 2.7.3 Call Z100: 2.7.2 Actual parameters
	Macro call	Z100: 4.2.3
	Create request	Z100: 2.7.2
	Transition option	Z100: 4.3.4 Transition option Z100: 5.3.1.9.1 Range condition

Symbol Appearance	Symbol Name	References to Z.100
	Continuous signal or enabling condition	Z100: 4.11 Continuous signal Z100: 4.12 Enabling condition
	Start	Z100: 2.6.2 Start Z100: 6.3.3 Virtual transition
	Priority input	Z100: 4.10 Priority input Z100: 2.6.4 Stimulus Z100: 6.3.3 Virtual transition z100: 5.4.3 Variable z100: 5.4.3.1 Indexed variable z100: 5.4.3.2 Field variable
	Stop	Z100: 2.5.8.2.3
	Procedure start	Z100: 2.4.6
	Inlet	Z100: 4.2.2
	Return	Z100: 2.6.8.2.8
	Outlet	Z100: 4.2.2
	Procedure reference	Z100: 2.4.6 Procedure definition

Symbol Appearance	Symbol Name	References to Z.100
	Operator reference (an SDL/GR extension defined in the SDL Suite)	Z100: 5.3.2 Referenced operator in SDL/PR
	Gate	Z100: 6.1.4 Gate Z100: 2.5.5 Signal list

Note:

The operator reference symbol is not part of the current Z.100 recommendation. It has been added to the SDL Editor as a convenience for the user. It makes operators visible in the SDL structure which is handled by the Organizer and facilitates thus navigating.

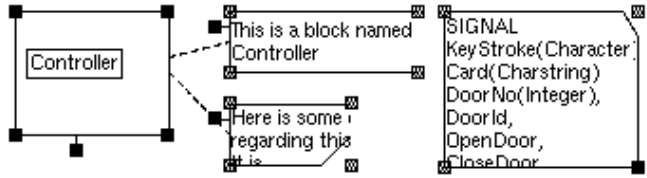
The operator reference symbol does not, however, refer to the operator diagram implicitly. References to operator diagrams must be explicitly entered in SDL/PR, as stated in Z.100.

Graphical Properties of Symbols

You can select and move all symbols that are available in the symbol menu, and you can assign the symbols arbitrary locations.

You can resize certain symbols; these are indicated by filled selection squares. Other symbols can only be partially resized or cannot be resized at all[1]; this is shown by grayed selection squares.

1. The size of non-resizeable symbols is computed by the SDL Editor to fit the text contained in the symbol.



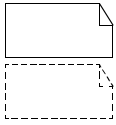
This symbol can be resized...

...while that one can be either full sized or clipped...

...and the third one can be resized by dragging the lower right corner.

Figure 374: Resizeable and non-resizeable symbol

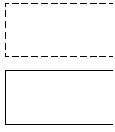
No symbol is allowed to overlap any other symbol except the text symbol and the additional heading symbol.



Text / Additional Heading / Package Reference Symbols

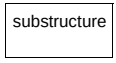
The text, additional heading and package reference symbols look like a piece of paper. When all of the text within a text symbol is in view, the upper right corner is “folded” down. When any portion of the text within a text symbol cannot be seen (because the text symbol is too small), the upper right corner is clipped or diagonally cut off. See Figure 374.

Clipped text symbols are printed in their whole on a separate sheet of paper.



Comment / Text Extension Symbols

The comment and text extension symbols can be shown either in full size enclosing the complete text or clipped (collapsed) where the symbol is in the minimum size and only the first part of the text is visible. The clipped symbol is visualized as if the lower right corner has been cut off. See Figure 374.



Block Substructure Symbol

The appearance of this symbol differ from the drawing rules in Z.100. The non editable text “substructure” is drawn inside the symbol. Without this extra text it is impossible to distinguish the block symbol and the block substructure symbol by their visual appearance.

The appearance can be set to be according to Z.100 by the Editor preferences ScreenZ100Symbols and PrintZ100Symbols. Setting these to on means that the extra text will not be drawn or printed.

Symbol Text Attributes

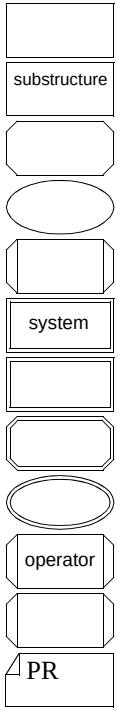
Most SDL symbols have one or multiple text attributes. A text attribute should be filled with an SDL/PR expression (textual expression) that is syntactically correct according to Z.100. Depending on the correctness of the text, the SDL Suite tool set has the ability to perform the following operations:

- Syntactic Analysis
- Semantic Analysis
- Pretty-printing
- Generation of reports
- Generation of code.

Syntax Checking on Text Attributes

You are not forced by the SDL Editor to fill text attributes with text. However, a context sensitive syntax check is performed for each text attribute. The first located syntax error is indicated by a red bar underlining the text where the error occurs.

A global syntax check for the complete diagram is performed by the SDL Analyzer.



Reference Symbols

Diagram reference symbols allow to build an entire SDL system by referring to diagrams that are structurally related. Reference symbols are the following:

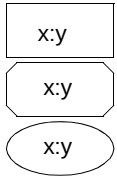
- Block reference
- Process reference
- Block substructure reference
- Service reference
- System type
- Block type
- Process type
- Service type
- Operator reference
- Procedure reference
- Text Reference

These symbols are handled by the SDL Editor in order to ensure consistency between the SDL diagrams and the SDL structure that is handled by the Organizer. Because of that, the SDL Editor imposes the restriction that a reference symbol must be unique within an SDL diagram (this restriction is not in conflict with Z.100).

You are permitted to have diagram reference symbols that are not assigned any name.

The SDL Editor performs a number of checks when you edit an SDL reference symbol. Furthermore, when you double-click a reference symbol, the SDL diagram that the symbol refers to will be opened.

See also "Working with Diagram Reference Symbols" on page 1901.



Instantiation Symbols

You may use any of the following reference symbols for the instantiation of a type:

- Block reference
- Process reference
- Service reference.

An instantiation symbol differs from a “normal” reference symbol in the sense that the syntax of the symbol’s text differs. The syntax is:

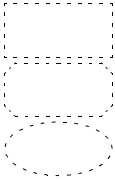
```
a:a_type
```

where *a_type* is the name of the SDL type diagram (block type, process type or service type diagram) that is instantiated into the diagram with the name *a*.

Instantiation symbols are visualized in the Organizer structure. The Organizer structure is updated to reflect the nature of the changes applied to the SDL diagram, in a similar fashion as for diagram reference symbols (see “Reference Symbols” on page 1893).

The names of instantiation symbols must be unique within an SDL diagram.

See also “Working with Diagram Instantiation Symbols” on page 1901.



Dashed Reference Symbols

Any of the following reference symbols may be dashed:

- Block reference
- Process reference
- Service reference.

Dashed symbols are visualized in the Organizer structure. The Organizer structure is updated to reflect the nature of the changes applied to the SDL diagram, in a similar fashion as for diagram reference symbols (see “Reference Symbols” on page 1893).

The names of dashed reference symbols must be unique within an SDL diagram.

See also “Working with Dashed Symbols” on page 1902.

Lines

Lines are the graphical objects that interconnect symbols. One line only is available in the symbol box, namely the gate, which is handled as a symbol. You can insert it by selecting it and placing it into the drawing area.

You insert the other lines by selecting a symbol and dragging the *handle* that appears on the source symbol and connecting it to the target symbol. (Some symbols have multiple handles).

Lines are always connected to symbols, they are not allowed to exist on their own.

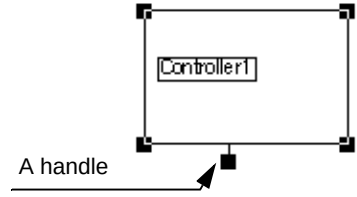


Figure 375: A handle

You can select, move and reshape lines. Some layout work is performed automatically by the SDL Editor.

A line is allowed to overlap any other object.

Lines in Interaction Diagrams

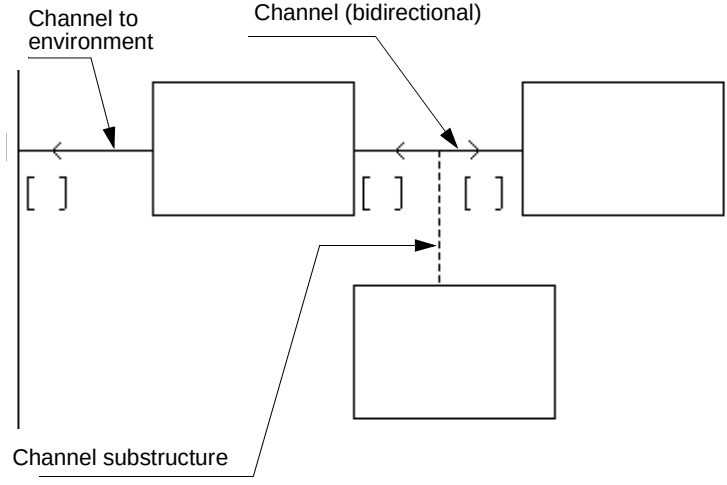


Figure 376: Lines (1)

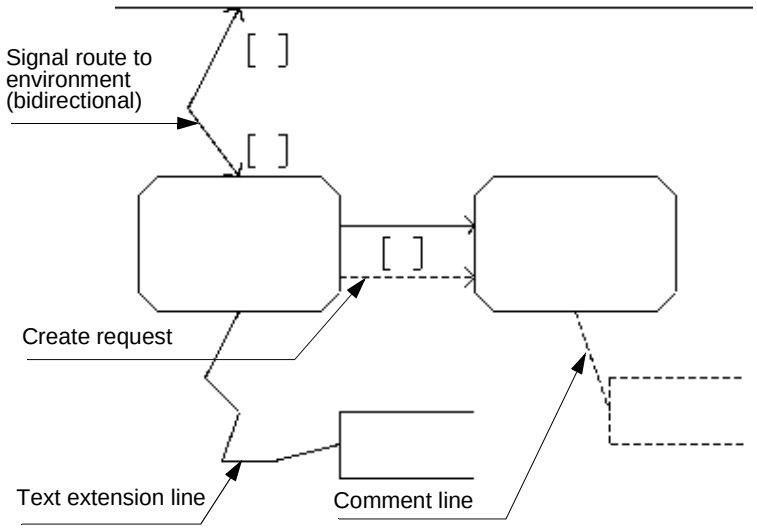


Figure 377: Lines (2)

The following lines are defined in interaction diagrams (see Figure 376 and Figure 377):

- channels
- channel substructures
- signal routes
- create requests
- text extension lines
- comment lines

You can reshape each of these lines and you can also move the connection points.

Lines in Flow Diagrams

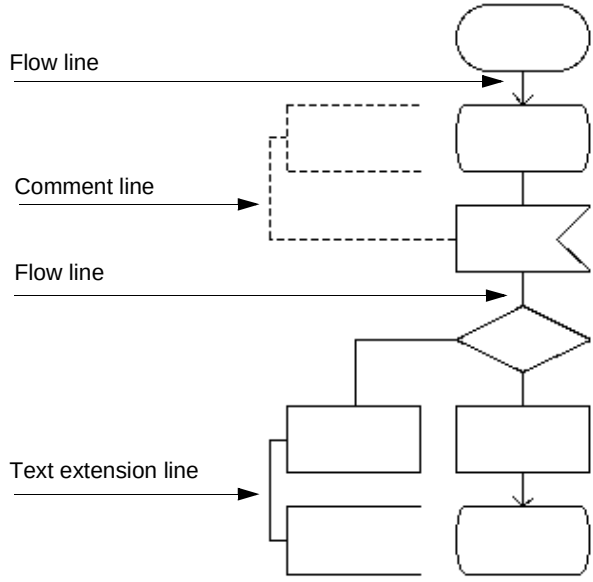


Figure 378: Lines (3)

On flow diagrams, the following lines are defined (see Figure 378):

- flow lines
- comment lines
- text extension lines

Line segments on flow pages always use a 90-degree angle. You can add segments on a line, but connection points are fixed.

Syntax Checking on Lines

The SDL Editor checks that the target symbol is in accordance with the syntactic rules imposed by SDL. If not, the SDL Editor will refuse to connect the symbol. When you edit the text attributes are instantly checked for syntactical errors.

Textual Objects

Textual objects are the textual attributes that are related to a symbol or a line. Each of these attributes is prepared by the SDL Editor – you need of course to fill in their textual contents.

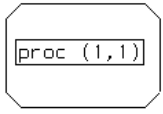
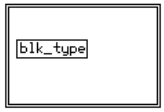
Textual attributes are indicated by a small rectangle which appears upon selection of the symbol or line the attribute belongs to.

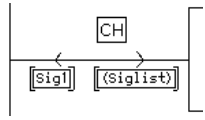
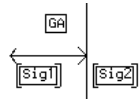
You can select and edit textual objects. You can move them freely, as long as their location does not violate the rules defined by Z.100.

Textual objects are allowed to overlap any other objects.

Textual Objects in Interaction Diagrams

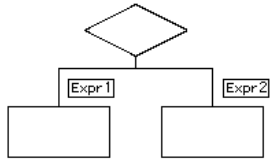
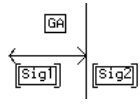
The following textual objects are defined in interaction diagrams:

Graphical Appearance	Textual objects
	• Name of a diagram reference symbol – Includes the *number of instances* for a process reference symbol
	• Name of a diagram type reference symbol
	• Name of a diagram instantiation symbol (e.g. `inst:blk_type`) – Names of graphical connection points in diagram instantiation symbol (e.g. `CON1`)

Graphical Appearance	Textual objects
	• Name of a channel or signal route (e.g. `CH`) – Signal list on a channel / signal route (there are 2 signal lists if the line is bidirectional, e.g. `Sig1` and `(Siglist)`) – Connection point for channel / signal route to / from environment, e.g. `CON`
	• Name of a gate, e.g. `GA` – Signal lists on a gate (there are 2 signal lists when the gate is bidirectional, e.g. `Sig1` and `Sig2`)

Textual Objects in Flow Diagrams

In flow pages, the following textual attributes are defined:

Graphical Appearance	Textual objects
	• Decision expressions connected to a decision symbol or transition option, e.g. `Expr1` and `Expr2`.
	• Name of a gate, e.g. `GA` – Signal lists on a gate (there are 2 signal lists when the gate is bidirectional, e.g. `Sig1` and `Sig2`

Graphical Connection Points

Connection points are text objects which are created automatically by the SDL Editor when you draw a channel or signal route to the frame or to an instantiation symbol.

They are handled in a similar way as other text attributes. You are free to fill connection points on the frame or not. The alternative is to define textual connection statements between channels and signal routes in a text symbol.

Change Bars

A change bar is a vertical line to the left of a text in an SDL diagram. It visually identifies changes in the SDL system.

A solid change bar indicates that the text has been changed. A dotted change bar indicates that a text or its associated symbol has been moved.

The change bars associated with the diagram name and the page name act as change bars for the complete page. As soon as a change has been made to a page, change bars for the diagram name and the page name are added.

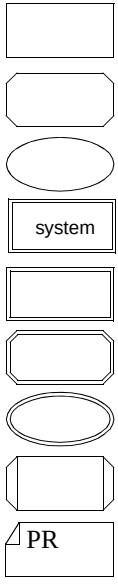
You can turn change bars on and off and also clear them from the Organizer. See "Change Bars" on page 147 in chapter 2, *The Organizer*.

Working with Symbols

The main operations provided by the SDL Editor on symbols is described in this section.

- *Working with Diagram Reference Symbols*
- *Working with Diagram Instantiation Symbols*
- *Working with Dashed Symbols*
- *Selecting Symbols*
- *Adding Symbols*
- *Inserting a Symbol into a Flow Branch*
- *Navigating in Flow Diagrams*
- *Navigating from Symbols*
- *Cutting, Copying and Pasting Symbols*
- *Moving Symbols*
- *Resizing Symbols*
- *Mirror Flipping Symbols*
- *Adjusting Symbols to the Grid*
- *Editing the Diagram Kernel Heading*.

In addition, text editing functions are provided for the text associated with the symbols.



Working with Diagram Reference Symbols

When you perform certain modifications to the diagram reference symbols listed in the left margin (*system type*, *block*, *block type*, *substructure*, *service*, *service type*, *process*, *process type*, *procedure and text reference symbols*), the SDL Editor checks with the Organizer to verify that the modifications are valid. This occurs in the following situations:

- *Adding a Diagram Reference Symbol*
- *Adding an Operator Diagram Reference Symbol*
- *Renaming a Diagram Reference Symbol*
- *Removing Symbols*

Working with Diagram Instantiation Symbols

Instantiating a Diagram Type

The implication of an instantiation symbol is that the description of the type diagram is actually instantiated, i.e. a physical copy is created. This

copy may then be given additional properties using SDL's inheritance and specialization mechanisms.

When a diagram type is to be *instantiated*, do as follows:

1. Make sure the type diagram is visible according to the SDL scope rules.
2. Insert a diagram reference symbol. When specifying the name of the symbol, use the syntax `<instance_name>:<type_name>`.
3. When the symbol is deselected, the diagram structure is immediately updated to show the instantiation (this is only true if the *Instance Diagram* option is on in the Organizer's *View Options*).

Working with Dashed Symbols

Dashing and Undashing a Reference Symbol

Dashed reference symbols are used to refer to an object that is defined elsewhere in one of the supertypes of the current subtype.

To *dash* a reference symbol:

1. Select the reference symbol.
2. Select *Dash* from the *Edit* menu.

Note that a symbol cannot be dashed if there are syntax errors in the name attached to the symbol. Also if a syntax error is introduced in a dashed symbol the symbol will automatically be undashed.

Selecting Symbols

With the SDL Editor, you often perform actions that apply on a selection. This section discusses topics related to selection of symbols.

Selecting a Flow of Symbols

You often want to select a flow of subsequent symbols in a flow page, including all branches - the *tail* of the flow. To select a flow of symbols:

1. Click on the symbol where the flow starts.
2. From the *Edit* menu point to *Select Tail*. The tail of the flow is selected. See Figure 379, where the left picture shows only one select-

ed object and its associated connections, whereas the right picture shows the selected symbol and its connections plus the selected rest of the flow diagram – the *Selected Tail*.

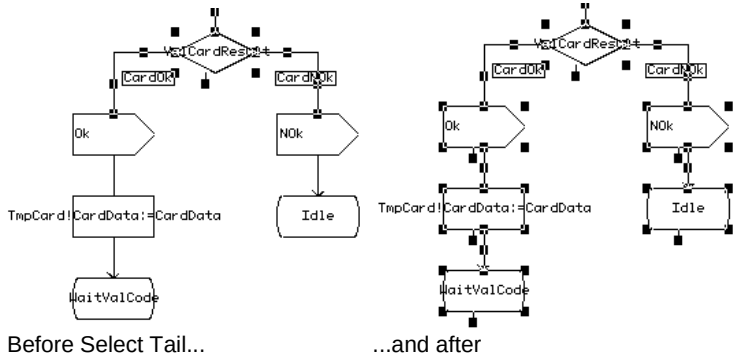


Figure 379: Selecting the tail of a subtree

Selecting a Symbol That Has Associated Objects

Consider the following when you select a symbol which has lines or text associated with it:

- When you select a symbol, you implicitly select the associated objects as well. A text cursor is inserted directly in the text associated with a symbol, and the *text window* will be updated to contain the text.
- If you point to a line associated with a symbol, you will select the line, **not** the symbol. The process of selecting lines is described in "Selecting Lines" on page 1932.
- If you point to the text associated with a symbol, the resulting selection depends upon the type of the symbol.

Displaying Selected Symbol Attributes

Selected symbol attributes are displayed as follows:

- When you select a symbol from which you can draw lines, it is displayed with a *handle* which you use when drawing lines from that symbol. Handles are shown in Figure 380.

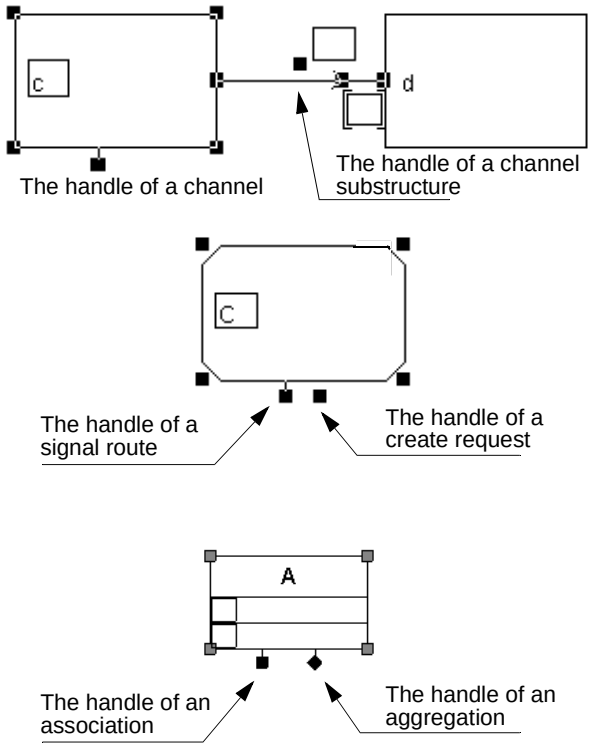


Figure 380: A selected symbol's handles

- *Text attributes* associated with a selected symbol are marked by a rectangle around the text, as shown in Figure 381. For class symbols, this only applies when text attributes contain no text.

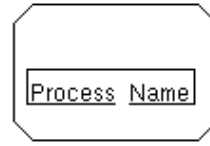


Figure 381: A selected symbol's text attributes

- In an interaction page, a line is marked by drawing filled *selection squares* at the line's starting point, ending point, and each breakpoint. In the middle of each line segment is a tiny selection square that provides an easy way to create new breakpoints. See Figure 382.

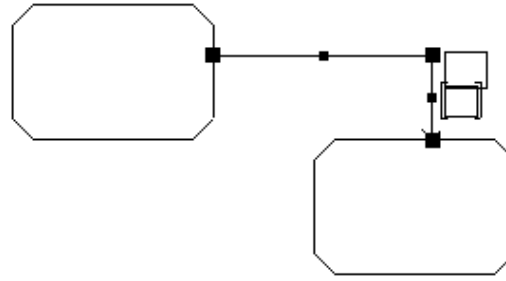


Figure 382: The selection squares of a line in interaction pages

- In a flow page, a selected flow line is marked by drawing filled *selection squares* in the middle of each line segment. This indicates that you can move the individual line segments (but not the flow line as a whole). See Figure 383.

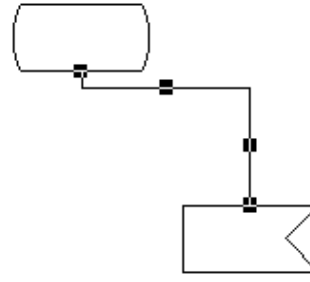


Figure 383: The selection squares of a line in flow pages

Adding Symbols

You may place symbols into the drawing area in either manual mode or in automatic mode.

Adding Symbols in Manual Mode

This section describes how to do this in manual mode. The automatic mode is described in "Adding Symbols in Automatic Mode" on page 1908.

Selecting and Placing a Symbol

1. Click the symbol in the symbol menu.
2. Click at the desired location in the drawing area. Overlap of symbols is not allowed.

   The symbol is drawn and adjusted to the symbol grid.

Entering Symbol Text Attributes

Once a symbol has been added, you can enter the text into the symbol's text attributes. The symbol remains selected. Enter the text required directly using the keyboard or edit the text inside the *text window*.

Connecting the Symbol to Another Symbol

Once the symbol is at the desired location it can be connected to another symbol as required.

To connect two symbols:

> **Note:**
>
> The line will be drawn only when the SDL syntax rules allow the symbols to be interconnected.

1. Add the *source symbol*, i.e. the symbol from where you want to draw the line.
2. Point the mouse onto the handle of the source symbol, and start dragging the handle. As soon as mouse motion has begun, you can release the mouse button. Point to the *target symbol* and click the mouse button. The line is drawn to the symbol and required text attributes are added to the connecting line. The box without brackets

is reserved for the name of the channel, while the box with brackets [ ] is for the signal list.

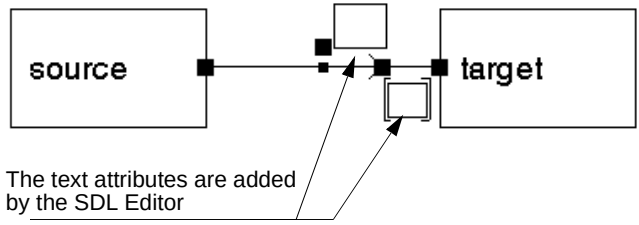


Figure 384: Having connected two symbols

3. Fill in the text attributes.

Conversely, once you have placed a symbol in the drawing area, you can follow the same procedure from a symbol resident in the drawing area to the new symbol.

Connecting the Symbol to the Environment

1. Hold the mouse pointer on the handle of the symbol, drag it to the frame (environment) and release the mouse button.
2. Enter text in the channel or signal route and connection point boxes.

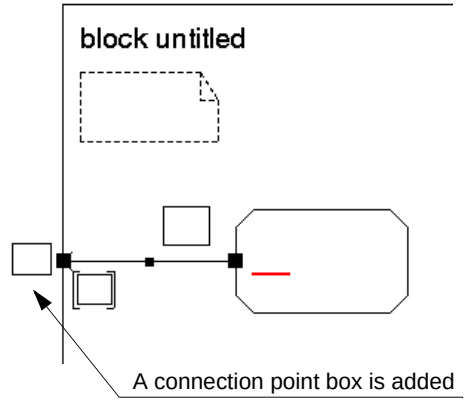


Figure 385: Having connected the symbol to the environment

Adding Symbols in Automatic Mode

The automatic mode is particularly useful when you want to add symbols sequentially or in parallel in a symbol flow. Both situations are described below.

Adding in Sequence

To add symbols sequentially within a flow page, proceed as follows:

1. Select the source symbol in the drawing area.
2. Double-click the *target symbol* in the *symbol menu.*

   The target symbol is added to the source symbol, and a flow line is drawn between the two symbols if the syntax rules allow it.
3. Fill in the text in the target symbol.

The target symbol is positioned directly beneath the source symbol if possible. Otherwise, the added symbol is placed in the first available location to the right.

After this operation, the target symbol is automatically selected and the source symbol is de-selected.

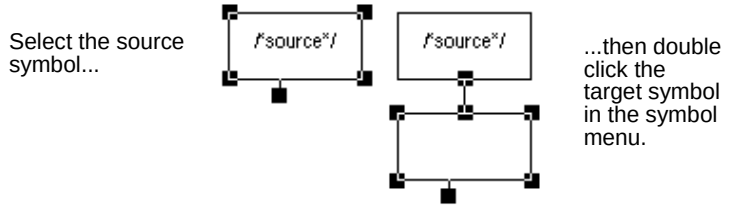


Figure 386: Adding in sequence

Adding in Parallel

To add symbols in parallel in a flow page:

1. Select the *source symbol* in the drawing area.
2. Press `<Shift>` and double-click the *target symbol* in the *symbol menu*.

   The target symbol is added to the selected symbol. The source symbol remains selected.

You can repeat this procedure in order to add additional symbols to the selected symbol. The target is positioned immediately below the source symbol, if possible. Otherwise, the target symbol is placed in the first available location to the right.

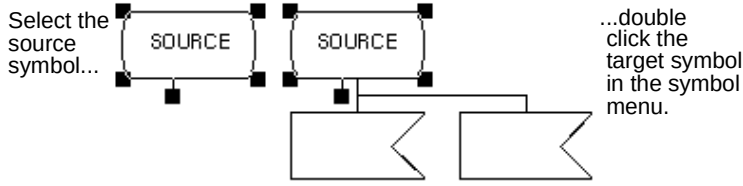


Figure 387: Adding in parallel

Using Default Placement

You can also take advantage of the auto-placement by double-clicking a symbol without having to connect it to other symbols.

Insufficient Space Left

When you insert a symbol using the double-click facility, there may not be any space left to perform the operation, The SDL Editor informs you of this with a message.

You have the following options:

- Click *OK* to increase the page. The *Drawing Size* dialog will be opened, where you can specify the new size of the SDL Page.

  If you select this method, the SDL page may not fit any longer into the paper that you have set up with the Print preferences. The result may be an SDL page that requires multiple sheets of paper when printing it (unless you rescale it).

- To *Cancel* the operation. Try to find a more suitable location for the new symbol and use manual placement (see "Adding Symbols in Manual Mode" on page 1906).

Adding a Gate Symbol

The gate symbol differs slightly from other symbols when you add it.

Adding a Gate in Manual Mode

1. Select the gate symbol.
2. Move the pointer into the drawing area. The gate is immediately displayed and connected in a 90-degree angle to the frame and the environment.
3. Move the mouse to a suitable location. The gate symbol follows the mouse motion and is automatically reconnected.
4. Click the mouse.
5. Enter the name of the gate and the signals it conveys.

> **Note: Coalesced Gates and Connection Points**
>
> See "Connecting a Gate Symbol with a Channel or Signal Route" on page 1919 if you get a message with the text "A Gate has coalesced with a connection point...".

Adding a Gate in Automatic Mode

You may add and connect a gate to a channel or signal route with a double-click. The channel or signal route must not be already connected to a gate or connection point.

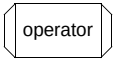
1. Select the channel or the signal route.
2. Double-click the gate symbol in the symbol menu.
3. If required, redirect or bidirect the gate so that it matches the direction of the channel or signal route.
4. Enter the name of the gate and the signals it conveys.

Adding a Diagram Reference Symbol

When you add a diagram reference symbol, the SDL Editor checks that the name and type are unique within the SDL diagram. The SDL Editor will refuse any duplicates. The newly added reference symbol is then inserted into the Organizer's diagram structure.

1. Select the symbol to be added from the symbol menu and add it to the diagram.
2. Type in the name of the symbol.
3. Deselect the symbol. The SDL Editor checks that the name and type are unique within the SDL diagram being edited.

   If the new reference symbol is accepted, it is inserted into the diagram structure, located beneath the referring diagram.

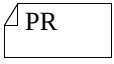


Adding an Operator Diagram Reference Symbol

In a data type construct it can be stated that an operator diagram is referenced elsewhere. This means that the operator will not be described in the current place, but in an operator diagram.

The SDL Editor does not know that the operator is referenced in a data type. The operator reference symbol is available for this situation. It is a convenience that has been added to the SDL Suite tool set to facilitate navigating within an SDL system. Its presence in an SDL diagram does not modify the meaning of the diagram.

You add an operator reference symbol in a similar manner as when adding any other reference symbol. See "Adding a Diagram Reference Symbol" on page 1911.



Adding a Text Reference Symbol

The Text reference symbol is used to include SDL PR into the SDL system.

You add an operator reference symbol in a similar manner as when adding any other reference symbol. See "Adding a Diagram Reference Symbol" on page 1911.

Inserting a Symbol into a Flow Branch

The SDL Editor allows you to insert symbols into a flow branch. When inserting symbols, the SDL Editor will rearrange the page in order to prepare the required space for inserting the symbol. This can be done in different ways:

- By pushing the tail of the branch downwards. This is the preferred method.
- By increasing the size of the SDL page.

Inserting a Symbol into a Flow Branch

1. Select the symbol in the symbol menu.
2. Point to the line interconnecting the symbols where the new symbol is to be inserted.
3. Click the mouse.

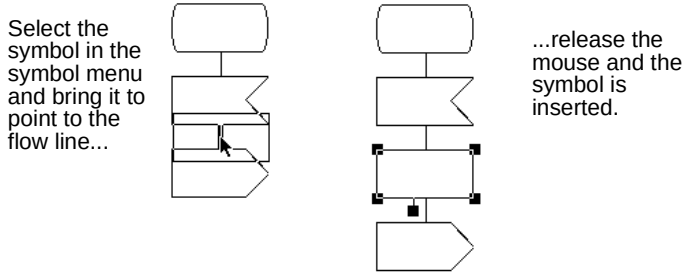


Figure 388: Inserting a symbol into a flow branch

Inserting a Symbol by Double-Click

To insert a symbol into a flow branch by double-clicking it:

1. Select the line that interconnects the two symbols. Alternatively you can select the symbol preceding the line, see "Insert Paste" on page 2007.
2. In the symbol menu, double-click the symbol to be inserted.

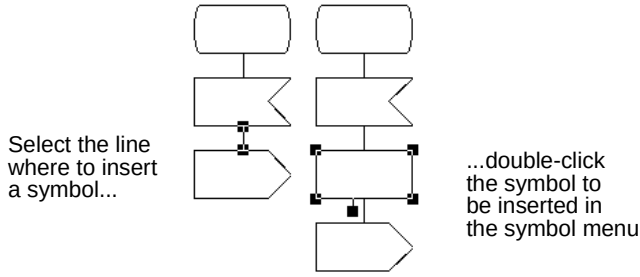


Figure 389: Inserting a symbol with double-click

Navigating in Flow Diagrams

For certain symbols you can use the *Navigate* command in the *Tools* menu or double click on the symbol to navigate around in the diagrams. These symbols are:

- Create request, procedure call and macro call
- State and nextstate
- Inconnector and outconnector
- Reference symbols; see further "Navigating from a Diagram Reference Symbol" on page 1914

> **Note: Navigating from a Class Symbol**
>
> Navigating from a class symbol is possible only by using *Tools>Navigate*. See further "Navigating from Class Symbols" on page 1915

Navigating from Create Request and Procedure/Macro Call

For a create request, procedure call or macro call the *Navigate* command will show the corresponding process, procedure or macro diagram. The diagram is found by searching in the Organizer structure for all relevant diagram types matching the name used in the symbol. If one and only one match is found this diagram will be shown in the editor. If there exists more than one diagram a dialog is issued.

Navigate to the desired diagram by selecting it in the dialog.

Navigating from State/Nextstate

For a state or nextstate the *Navigate* command will show other occurrences of the same name(s) in other state or nextstate symbols within the same diagram. If only one symbol matches the name search this symbol will be shown. If more than one symbol have the same name, a dialog is shown.

Navigating from Connectors

For an outconnector, the *Navigate* command will show the corresponding inconnector in the same diagram having the same name as the outconnector.

Similarly for an inconnector; you can navigate to all occurrences of outconnectors within the same diagram having the same name as the inconnector.

Navigating from Symbols

Navigating from a Diagram Reference Symbol

When you double-click or choose the Navigate command on a diagram reference symbol, the SDL diagram that the reference symbol refers to is opened, provided that the diagram is included in the Organizer structure.

This feature allows easy navigation down an SDL hierarchy.

See also "Navigating from an Instantiation Symbol" on page 1914 and "Navigating from a Dashed Symbol" on page 1915.

Navigating from an Instantiation Symbol

You may use instantiation symbols to navigate in the diagram structure.

When you choose *Navigate* in the *Tools* menu or double-click on an instantiation symbol, the SDL Editor matches the name of the type referred to in the instantiation symbol against the existing type diagrams in the Organizer structure. The following happens depending on how many type diagrams that match the given diagram name.

- **One and only one diagram is found**: This diagram will be shown.

- **No diagram matches the name**: A dialog will appear which offers the possibility to open the Type Viewer window which can be used to navigate among the inheritance and redefinition trees extracted from the SDL system:
- **More than one diagram is found**: A dialog will appear showing the file names of all the found diagrams. Additionally the possibility to show the Type Viewer window is available.

Navigating from a Dashed Symbol

When you select *Navigate* in the *Tools* menu or double-click a dashed symbol, a scenario similar to the action of navigating from an instantiation symbol takes place. See "Navigating from an Instantiation Symbol" on page 1914

Navigating from Class Symbols

For a class symbol, the *Navigate* command will show other occurrences of the same name(s) in other class symbols within the same diagram. If only one symbol matches the name search, this symbol will be shown. If several symbols have the same name, a select dialog is shown.

Cutting, Copying and Pasting Symbols

The clipboard of the SDL Editor allows you to cut or copy a selection of symbols. The selection may then be pasted to:

- The same page
- Pages of the same diagram, provided the page types match
- Other diagrams, provided the diagram types match

See also "Cutting, Copying and Pasting Reference Symbols" on page 1917.

Cutting and Pasting Symbol(s)

A dialog may appear when cutting or pasting an object with link endpoints. See "Deleting an Object" on page 461 in chapter 9, *Implinks and Endpoints* or "Pasting an Object" on page 461 in chapter 9, *Implinks and Endpoints*.

Limitations when Pasting

> **Note: Effect of Syntax Check when Pasting**
>
> Some symbols may not be pasted if the syntax check is on. The clipboard buffer content must comply with the SDL syntax rules. Any **symbols which do not comply will be omitted**. (The clipboard buffer content may have been copied from a page edited with syntax checking turned off.)

Pasting is not always possible. Note the following cases, and refer to "Relationship between Diagrams and Pages" on page 1848.

- The selection cannot be pasted if the page is too small. In this case, you need to enlarge the page. See "Resizing a Page" on page 1958.
- You are not allowed to paste symbols so that they overlap existing symbols in the drawing area. If you attempt to do this, the *Paste* operation is cancelled. Try another suitable location.
- You are not allowed to paste flow symbols into an interaction page and vice versa. In this situation, the *Paste* menu choice is dimmed.

Pasting on Flow Pages

On a flow page, symbols are pasted along with the flow lines that interconnect them.

Pasting on Interaction Pages

On an interaction page, symbols are pasted along with the lines that interconnect them. Any lines that connect the pasted symbols to the environment will also be drawn.

Pasting Symbols into a Flow Branch

On a flow page, symbols can be pasted into a flow branch in a similar manner as "Inserting a Symbol into a Flow Branch" on page 1912.

1. Copy or Cut one or more symbols in a flow page. The symbols must be connected and placed below each other.
2. Go to the page where you want to insert the symbols and select a symbol or a flowline.
3. Select *Insert Paste* from the *Edit* menu.

4. The pasted symbols will be placed below the selected symbol and inserted into the flow branch. Any previously following symbols will be automatically shifted downwards to make place for the inserted symbols.

Cutting, Copying and Pasting Reference Symbols

When you cut or copy a reference symbol, the possible sub-hierarchy shown in the Organizer below this reference symbol is also placed in the clipboard. When pasted in the same diagram or in another diagram this sub-hierarchy will be restored at the new position.

> **Note:**
>
> When a reference symbol is cut, the sub-hierarchy that is shown in the Organizer will be removed. However, the hierarchy will be restored if you select *Paste* or *Undo*.

When you copy or paste a diagram reference, it is assigned a new name that consists of the original name, preceded with the prefix *CopyOf*. The prefix is repeated as many times as necessary in order to build a name / type combination that is unique within the SDL diagram being edited. If the cut/copied symbol had an underlying structure this structure is restored after the paste.

Cutting, Copying and Pasting Class Symbols

When cutting, copying or pasting a class symbol, one or more operators with a defined body code may be completely removed, i.e. there is no longer any class symbol that refers to a specific operator. If so, the following dialog is shown:

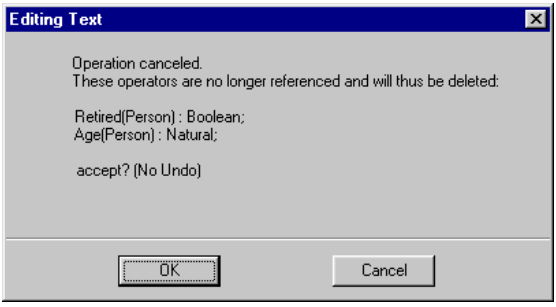


Figure 390: Checking with user if OK to remove operators

You have the following options:

- Accept the deletion of the listed operators and their defined body code by pressing *OK*.
- Cancel the issued operation by pressing *Cancel* and try to correct the error.

Caution!

By confirming the deletion of the operators, you accept that the listed operators, and possibly defined body codes, will be permanently removed. No *Undo* option is available.

Moving Symbols

You may move one or more symbols in a single operation. The lines and text objects that are related to the symbol will be moved accordingly in order to preserve as much as possible of the original appearance. When you move a symbol, its position is automatically adjusted to fit exactly into the symbol grid.

Symbols are not allowed to be moved to positions where they would overlap other symbols.

Moving Symbols and Text Attributes

You can move symbols and text attributes by pointing and dragging to the desired position. To grab the text attribute the pointer must be at the border of the text extent.

You cannot move text within flow symbols and class symbols.

Connecting a Gate Symbol with a Channel or Signal Route

When you move a gate symbol in order to reconnect it to a line or vice versa (channel or signal route that is connected to a connection point), the following takes place:

- If the connection point is not assigned any name, the SDL Editor connects the gate symbol and the line.
- If the connection point is assigned a name, a conflict occurs, since the line cannot be connected to the connection point and the gate simultaneously. The following dialog is issued:

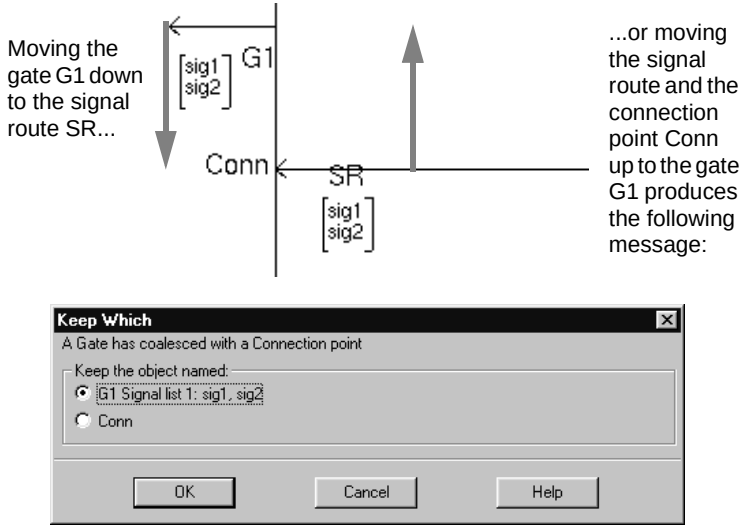


Figure 391: Prompting to solve connection conflict

You have the following options:

– To keep the gate G1 and to connect the line to it. The connection point Conn is removed.

– To keep the connection point Conn and to connect the line to it. The gate G1 is removed.

- Click *OK* to reconnect the line and symbol accordingly.

Note:

The situation above also occurs if you add a new gate symbol and place it on a connection point.

Resizing Symbols

The procedure for resizing a symbol differs slightly depending on whether the symbol is a flow page symbol or an interaction page symbol. When you resize a symbol, its size is automatically adjusted to fit exactly into the symbol grid. Any lines connected to the symbol are similarly adjusted to preserve as much as possible of the original appearance.

The text, the additional heading, the package reference, comment, class, and text extension symbols are handled differently. Their size is determined by the SDL Editor to fit the text they contain.

Specifying Default Size of Symbols

You define the default sizes for symbols in the Preference Manager:

- For interaction pages, the predefined size of a symbol is 30 * 20 mm.
- For flow pages, the predefined size of a symbol is 20 * 10 mm. The width / size relationship is always 2:1.

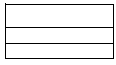
Resizing Interaction Page Symbols

1. Point to a corner of the rectangle that delimits the symbol. From this point on, the corner that you are pointing to is movable, while the opposite corner is fixed.
2. Drag the corner until the symbol has the desired size. An interaction symbol cannot be adjusted to a size less than 15*10 mm.

3. Release the mouse button. The symbol is redrawn with its size adjusted to fit exactly into the symbol grid. The lines connected to a resized interaction symbol are automatically adjusted to preserve a nice appearance.

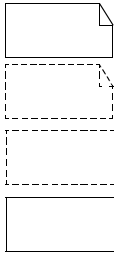
Resizing Flow Page Symbols.

1. Point to a corner of the rectangle that delimits the symbol. From this point on, if you pointed to any of the lower corners, the top of the symbol is fixed (and in a similar way, if you were pointing to any of the upper corners, the bottom of the symbol is fixed).
2. Drag the corner until the symbol has the desired size. The symbol shrinks or swells both horizontally and vertically, the proportions remaining 2:1. A flow page symbol cannot be adjusted to a size less than 20*10 mm.
3. Release the mouse button. The symbol is redrawn.



Resizing the Class Symbol

The class symbol contains three text boxes. Whenever the text in any of the boxes grows bigger than the symbol, the class symbol is automatically re-sized.



Resizing Text / Additional Heading / Package Reference / Comment / Text extension Symbols

The text, additional heading and package reference symbol look like a piece of paper. When all of the text within a text symbol is in view, the upper right corner is “folded” down. When any portion of the text within a text symbol cannot be seen (because the text symbol is too small), the upper right corner is clipped or diagonally cut off. See Figure 392 on page 1922.

Clipped text symbols are printed in their whole on a separate sheet of paper.

These symbols react in the same way for resizing. There are several ways you can resize these symbols:

Adjusting to a Specific Size (only for Text / Additional Heading / Package Reference)

1. Place the mouse pointer on the filled selection square at the bottom right corner of the symbol.
2. Holding the left mouse button down, make the symbol smaller by pushing to the left or larger by pulling the corner to the right.

Toggling Between Minimum and Maximum Size

You may rapidly toggle between a text symbol's maximum and minimum size. This can be accomplished in two ways:

- Select the symbol and choose the *Collapse/Expand* command in the *Edit* menu.
- Double-click on any part of the symbol.

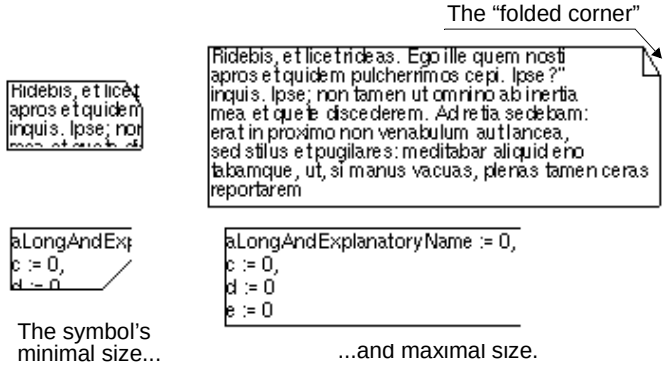


Figure 392: Minimizing and maximizing a symbol

Mirror Flipping Symbols

The input, priority input, output, comment and text extension symbols are drawn by default as displayed in the symbol menu. In some cases, the layout of the connecting lines would be nicer if the symbols were drawn flipped L/R.

1. Select the symbol.
2. Choose *Flip* from the *Edit* menu. The symbol is flipped L/R.

Flipping the text extension symbol mirrors it and redraws the connecting line

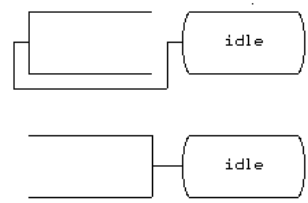


Figure 393: Flipping a symbol

Note:

By using the Preference Manager you can specify whether the output symbol in the symbol menu should point to the left or to the right, or you can have both. The same applies for the input and priority input symbols.

Adjusting Symbols to the Grid

The drawing area of the SDL Editor provides two grids for easier positioning of objects. These are the *symbol grid* and the *line grid*, which are invisible.

After you move or resize symbols, the position of the upper left corner of each symbol is positioned on the closest intersection in the symbol grid. In addition, resizing a symbol will adjust the size so that it matches the resolution of the grid, horizontally as well as vertically. The resolution of the symbol grid is variable and can be set with the Preference Manager. The default is 5 * 5 mm for interaction pages, and 25 mm * 15 mm for flow pages.

You can temporarily disable the grid by toggling the *Use Grid* to off in the *Diagrams Options* dialog.

Handling Symbols of Different Sizes

Since symbols adhere to the symbol grid, mixing symbols of different sizes may imply that symbols cannot be aligned symmetrically, and that enlarging symbols becomes impossible due to overlap of symbols already laid out.

You can align symbols of different sizes by disabling the symbol grid.

Editing the Diagram Kernel Heading

You can rename, retype and re-qualify a diagram with the SDL Editor, by modifying the diagram's kernel heading.

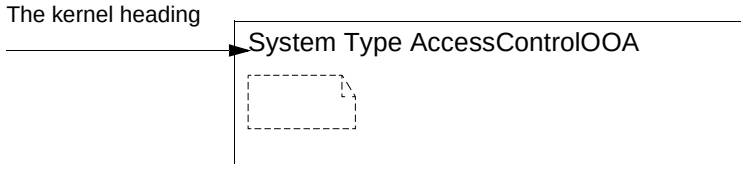


Figure 394: The kernel heading

See also "Transforming the Type of a Diagram" on page 1878 for more complicated transformation of diagram types.

The SDL Suite accepts a number of syntaxes for kernel headings, 13 in all. See "The Kernel Heading" on page 1858.

Note:

Changing the kernel heading of a diagram may have a large impact on the diagram's meaning. Also, the SDL diagram structure where the diagram is included may be affected to a large extent. You may, however, want to change the kernel heading in order to create a new diagram that you will use in a different context than the SDL structure it originally is tied to, or in order to perform a simple rename of diagrams.

Changing the Name of a Diagram

This is the bottom-up method you would use when renaming diagrams in an SDL structure.

1. Select the kernel heading.
2. In the text window, edit the kernel heading text.

3. Deselect the kernel heading. The SDL Editor checks that the name is still valid according to SDL rules. Any syntax error will cause a dialog with the following alternatives to be issued:

   – Clicking *OK*, you are returned to the text window where you should correct the error.

   – Clicking *Revert*, the kernel heading is reverted to its original contents.

   – Clicking *Ignore*, the erroneous kernel heading is accepted. The error will probably lead to other kind of error reports, typically when verifying the diagram with the Analyzer tool.

   The Organizer detects that the name of the diagram no longer matches the name that is expected in the diagram structure, and marks the diagram icon as mismatch. (A warning is also added to the Organizer Log window.)

4. To correct the mismatch, you have to update the name of the reference symbol accordingly. Use the command *Edit Reference Page* to bring the page with the reference symbol into view. Locate the reference symbol and rename it accordingly.

For renaming diagrams in a top-down fashion., see "Renaming a Diagram Reference Symbol" on page 1929.

Changing the Type of a Diagram

It is possible to re-type a diagram, provided the old and the new types are "compatible" from the SDL Editor's point of view. The basic compatibility criteria is that the old and new type of diagram must both be interaction diagrams or both be flow diagrams. Also checks are made on the pages that are contained in a diagram (some diagram types are allowed to have different types of pages in the same diagram, see "Relationship between Diagrams and Pages" on page 1848. The table below summarizes the type changes that are allowed.

case	A diagram of type...	...can be changed to any of:
1.	SYSTEM	SYSTEM TYPE BLOCK SUBSTRUCTURE BLOCK TYPE PACKAGE

case	A diagram of type...	...can be changed to any of:
2.	BLOCK	BLOCK TYPE SYSTEM SUBSTRUCTURE SYSTEM TYPE PACKAGE
3.	PROCESS	PROCESS TYPE SERVICE PROCEDURE SERVICE TYPE MACRODEFINITION OPERATOR
4.	SERVICE	SERVICE TYPE PROCESS PROCEDURE PROCESS TYPE MACRODEFINITION OPERATOR
5.	PROCEDURE	PROCESS SERVICE PROCESS TYPE SERVICE TYPE MACRODEFINITION OPERATOR
6.	SUBSTRUCTURE	SYSTEM BLOCK SYSTEM TYPE BLOCK TYPE PACKAGE
7.	MACRODEFINITION	PROCESS SERVICE PROCEDURE PROCESS TYPE SERVICE TYPE OPERATOR
8.	SYSTEM TYPE	SYSTEM BLOCK SUBSTRUCTURE BLOCK TYPE PACKAGE
9.	BLOCK TYPE	SYSTEM BLOCK SUBSTRUCTURE SYSTEM TYPE PACKAGE

case	A diagram of type...	...can be changed to any of:
10.	PROCESS TYPE	PROCESS SERVICE PROCEDURE SERVICE TYPE MACRODEFINITION OPERATOR
11.	SERVICE TYPE	SERVICE PROCESS PROCEDURE SERVICE TYPE MACRODEFINITION
12.	OPERATOR	PROCESS SERVICE PROCEDURE PROCESS TYPE SERVICE TYPE MACRODEFINITION
13.	PACKAGE	SYSTEM BLOCK SUBSTRUCTURE SYSTEM TYPE BLOCK TYPE

To change the type of a diagram:

1. Select the kernel heading.
2. In the text window, edit the kernel heading text to apply a new type. See the table above for allowed transformations.
3. Deselect the kernel heading. The SDL Editor checks that the type is still valid according to the table above. Any violation of these rules will cause a dialog with the following alternatives to be issued:
   - Clicking *OK*, you are returned to the text window where you should correct the error.
   - Clicking *Revert*, the kernel heading is reverted to its original contents.
   - Clicking *Ignore*, the erroneous kernel heading is accepted. The error will probably lead to other kind of error reports, typically when verifying the diagram with the SDL Analyzer.

Changing the Qualifier of a Diagram

By changing the qualifier that appears in the kernel heading, you can change the context of the diagram.

The SDL Suite accepts the two Z.100 notations for a qualifier, namely with and without the surrounding "<< >>".

Example 321: An SDL Qualifier ––––––––––––––––––––––––––––––

```
PROCESS TYPE <<SYSTEM MYSYS/BLOCK MYBLOCK>> Myproc
```

Alternatively:

```
PROCESS TYPE SYSTEM MYSYS/BLOCK MYBLOCK Myproc
```

––

To change the qualifier:

1. Select the kernel heading.
2. In the text window, edit the kernel heading text to apply the new qualifier.

3. Deselect the kernel heading. The SDL Editor checks that the new qualifier is syntactically correct. Any syntax error will cause a dialog with the following alternatives to be issued.

   – Clicking *OK*, you are returned to the text window where you should correct the error.

   – Clicking *Revert*, the kernel heading is reverted to its original contents.

   – Clicking *Ignore*, the erroneous kernel heading is accepted. The error will probably lead to other kind of error reports, typically when verifying the diagram with the SDL Analyzer.

Renaming a Diagram Reference Symbol

When you change the name of a diagram reference symbol, the SDL Editor checks that the new name is not in conflict with existing symbols. If the reference symbol passes this check, the Organizer structure is updated accordingly.

When renaming diagrams in a top-down fashion, you start by changing the name of a reference symbol. After the reference symbol is renamed, you should also update the referenced diagram's kernel heading accordingly to maintain consistency in the diagram structure.

To rename a diagram reference symbol and the referred diagrams:

1. Select the symbol to be renamed.
2. Change the name of the reference symbol.
3. Deselect the symbol. The SDL Editor checks that the name and type are still unique within the diagram.

   If the new name of the reference symbol is accepted, the diagram structure is updated to reflect the nature of the change.

4. To ensure that the change also applies to the referred diagram, you should use the Organizer command *Update Headings* (see "Update Headings" on page 100 in chapter 2, *The Organizer*). This command verifies and possibly changes the name in the diagram kernel heading against the expected name (which is the one that now is shown in the reference symbol). The mismatch between the name of the diagram is reported in a dialog:

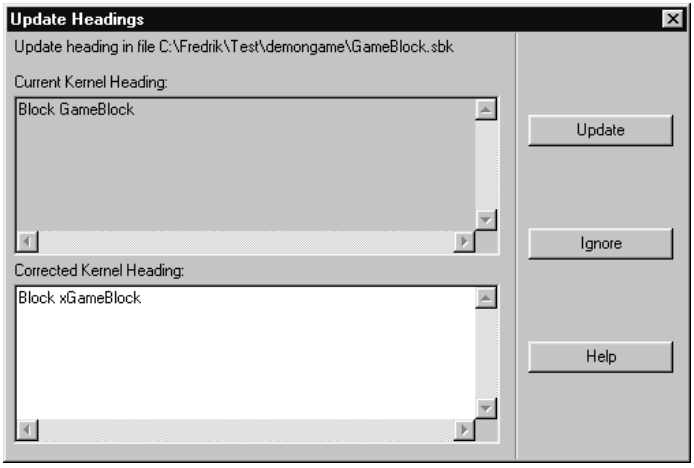


Figure 395: The mismatch in headings is reported

– Correct the mismatch by clicking *Update*.

Note that if you select not to update the kernel heading, you simply postpone the problem.

For renaming in a bottom-up way, see "Changing the Name of a Diagram" on page 1924.

Removing Symbols

When you remove a diagram reference symbol, the underlying structure is also removed from the Organizer structure. However, the hierarchy will be restored if the remove operation is restored by doing Undo.

1. Select the symbol(s) to be removed.
2. Select *Clear* from the *Edit* menu. The symbol(s) are removed, as well as the lines that were connected to these symbols.

Printing Symbols

A selection of symbols can be printed. This allows for instance to print only the portion of a diagram which is of interest.

1. Select the symbols to be printed.
2. Select *Print* from the *File* menu.
3. In the *Print Dialog*, click the *Setup* button. The Print Setup dialog will be issued. In the dialog, select the *Print only selected symbols* option.
4. Adjust, if necessary, other *Print options* and click the *OK* and *Print* buttons. See "Printing a Diagram" on page 1878 for more information.

Working with Lines

The main operations provided by the SDL Editor on lines, and their associated text attributes are:

- *Selecting Lines*
- *Drawing Lines*
- *Re-Routing and Reshaping Lines*
- *Redirecting and Bidirecting Lines*
- *Adjusting Lines to the Grid*

In addition, text editing functions are provided for the text associated with the lines.

Operations on lines are performed slightly different depending if you are editing

- *flow pages*
- *interaction pages*

The main difference between the two is that flow pages may only allow vertical and horizontal lines.

Some restrictions also apply when connecting symbols in flow pages, depending upon whether or not the syntax checking is enabled

Selecting Lines

You select lines and extend and reduce selections of lines with standardized selection commands.

Selecting a Line That Has Associated Objects

When you select a line, you implicitly select the associated attributes as well. To be able to edit a text attribute you must click on the text attribute and a text cursor will be inserted directly in the text, at the same time the text window will be updated to contain the text.

Displaying Selected Line Attributes

Selected line attributes are displayed as follows:

- Each text attribute is displayed inside a text selection rectangle.

- The following attributes are displayed with a filled selection square:
  - Each arrow (on interaction pages only)
  - Each line breakpoint (on interaction pages only)
  - The middle of each line segment (on interaction pages only a tiny selection square marks the handle to create a new breakpoint)
  - The middle of each line segment (on flow pages only)

Selecting Overlapping Objects

On a page, objects may overlap each other. A *layer order* is defined so that it is possible to determine how overlapping objects are stacked.

Drawing Lines

This section describes how to interconnect symbols with lines. Two modes are supported in the SDL Editor.

Drawing a Line with Layout Syntax Checking Enabled

The way that you draw a line differs somewhat depending upon whether you are working on an interaction page or a flow page. The differences will be pointed out later in this section.

When syntax checking is enabled, you are only allowed to draw lines that are correct according to the SDL rules. When syntax checking is disabled, the rules are less strict.

Designating the Start Point

1. Select the symbol from which the line will originate.
2. Point to the handle and drag it to begin forming the line. The line that is formed ends where the mouse pointer is located. Note that the mouse button can be released as soon as you have moved the mouse pointer.

Designating the End Point

1. Designate the start point. See previous section.
2. You can route the line and specify where it will end either automatically or manually.

- To route the line automatically:
  - Click any valid target symbol to specify where the line should end.

    A symbol is valid if the syntax rules enforced by the SDL Editor at edit-time allows the drawing of a line from the first symbol to the second symbol.
- To route the line manually, indicating where *line breakpoints* are to be inserted, follow the steps below.

  1. To enter manual mode, click any location in the drawing area other than a valid destination symbol.
  2. Click each location where you want to insert a line breakpoint.
  3. Click on a valid destination symbol.

     As long as you do not point to a valid destination symbol, the line will be routed from the selected symbol to the current mouse position. The automatic line drawing facility will determine the layout of the line until you have inserted the first breakpoint.

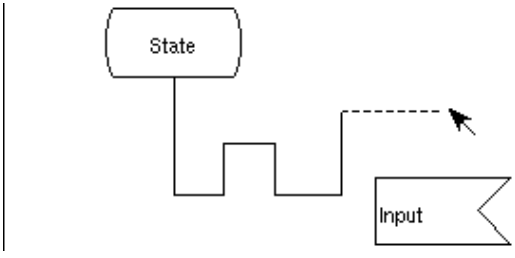


Figure 396: Drawing a line in manual mode

Drawing Multiple Lines

When you draw lines between symbols and keep the <Control> key pressed, the start point for the next line to draw is automatically transferred to the target symbol.

This feature provides the easiest way to draw multiple lines in rapid succession.

Differences between Flow and Interaction in Line Drawing

The way that you draw a line differs somewhat depending upon whether you are working on an interaction page or a flow page:

- On an interaction page, you can draw lines in all possible directions.
- On a flow page, you can only draw lines horizontally or vertically. The last line breakpoint, even if determined manually, will be adjusted so that the destination symbol is connected correctly, normally in the middle of the upper edge of the symbol. Exceptions are made for comment and text extension symbols, which are connected to the center of the right or left edge of the symbol.

Cancelling the Line Drawing Operation

You can cancel a line drawing operation in either of the following ways:

- Press <ESC>.
- Double-click any part of the drawing area that is not a valid destination symbol.

Drawing a Line with Layout Syntax Checking Disabled

The process of drawing a line is the same regardless of whether or not syntax checking is enabled. To enable and disable syntax checking, see "Turning Syntax Checking On and Off" on page 1852.

When drawing lines with syntax checking disabled, you may interconnect symbols in any way you want.

Re-Routing and Reshaping Lines

Reshaping a line refers to changing the path that the line follows. This operation does not affect the endpoints of the line. The flow lines only move in either a horizontal or vertical way.

Re-routing a line denotes changing the ending point of the line from one symbol to another symbol.

Reshaping a Flow Line

1. Point to one of the segments of the line.
2. Press the mouse and, holding down the mouse button, drag the segment to the required position.
3. Release the mouse button.

Re-Routing a Flow Line

You can move the ending point of a flow line, and reconnect it to any valid destination symbol. You can redraw the line either manually or automatically.

To move a line's ending point:

1. Point at the ending point that you want to reconnect.
2. Press the mouse. Start dragging the ending point. As soon as you move the mouse, the line's last segment follows the mouse motion and you can release the mouse button.
3. You can click additional *line breakpoints* where you like.
4. Click the destination symbol to complete the operation. If it is a valid symbol, the line will be re-routed automatically.

Re-Routing an Interaction Line

You can move any starting or endpoint to and from a symbol and the environment.

1. Point to the starting or ending point of the line.
2. Press the mouse and, holding the mouse down, drag it to the desired symbol.
3. If required, click to add line breakpoints at desired positions.
4. Terminate by clicking on the new symbol. The line is redrawn accordingly.

Moving an Interaction Line Breakpoint

To move a line breakpoint in an interaction page:

1. Point to the breakpoint that you want to move.
2. Press the mouse and drag the breakpoint to the desired location.
3. Release the mouse button to redraw the line.

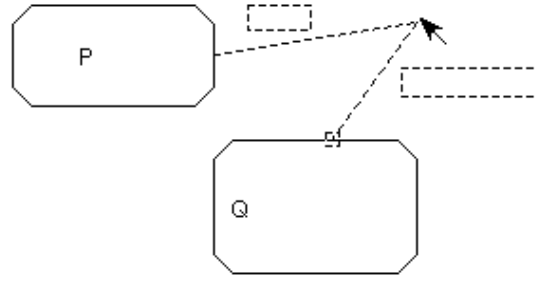


Figure 397: Moving an interaction line breakpoint

Inserting Breakpoints in an Interaction Line

1. Point at the middle of a line segment. When the line is selected a tiny selection square will be displayed at this point.
2. Press the mouse and drag it to shape the breakpoint.

An alternative way to insert breakpoints on an interaction line:

1. Press `<Ctrl>`
2. Point in a line segment where you want to insert a new breakpoint.
3. Press the mouse and drag it to shape the breakpoint.
4. Release the `<Ctrl>` key.

Moving an Interaction Line Segment

You can move a line segment in an interaction page as long as it is not a starting segment or ending segment.

1. Point to the line segment to be moved.
2. Press the mouse and drag the line segment to the desired location.

3. Release the mouse button. The line segment and the adjacent line segments are redrawn accordingly.

Moving Arrows

Arrows on channels, gates and signal routes denote the orientation of flow on that line. They are assigned default locations when creating the line.

You can manually move arrows **on channels** along the line.

1. Point to the arrow.
2. Press the mouse. Still holding the mouse down, drag it until the arrow reaches the new place.
3. Release the mouse. The arrow and the signal list are redrawn accordingly.

To create a non-delay channel you simply drag the arrow to the starting or ending breakpoint.

Moving a Line and Its Connection Point

A channel or a signal route may be connected to a connection point (usually located close to the line, immediately outside the frame).

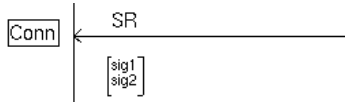


Figure 398: A signal route connected to a connection point.

Moving the line's ending point with the connection point unselected will move the line only and create a new connection point. The old connection point is removed.

To move the line **and** the connection point:

1. Click on the ending point of the line. Make sure the connection point is selected (i.e. the selection rectangle appears).
2. Drag the line's ending point to the new location. The connection point follows the mouse motion as well.
   – If the line's new ending point coincides with a gate symbol and the connection point has a name, a dialog is issued. The meaning of this message is described in "Connecting a Gate Symbol with a Channel or Signal Route" on page 1919.

Moving a Line Text Attribute

A line text attribute is a textual element that is related to a line. The following lines have associated text attributes:

- Gates
- Channels
- Signal routes
- Flow lines after decision, transition option and macro call symbols.

When drawing a line, the SDL Editor positions the text attribute close to the source symbol and originating line, so that it is easy to identify what line the text attribute is related to. You are however free to move the text attribute to any location in the drawing area within the frame.

To move a line text attribute:

1. Point to the border of the text attribute.
2. Press the mouse, and, still holding the mouse down, drag the text attribute to its new location.
3. Release the mouse button.

Redirecting and Bidirecting Lines

When you add a channel or signal route to an interaction diagram, the line is initially drawn in a uni-directional manner, originating from the source symbol and pointing to the target symbol.

The only way to draw a line from the environment to a symbol is to first draw it from the symbol and then redirect it.

Redirecting a Line

1. Select the line
2. Choose *Redirect* from the *Edit* menu. The line is redirected.

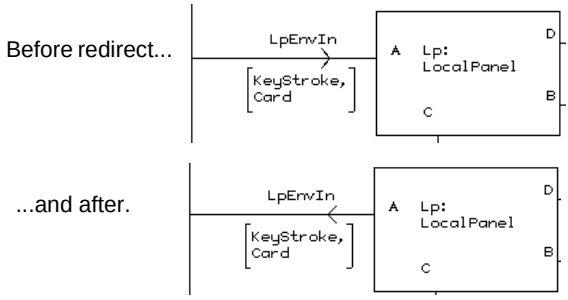


Figure 399: Redirecting a line

Bidirecting a Line

1. Select the line.
2. Choose *Bidirect* from the *Edit* menu. The line is bidirected.
3. Fill in the second signal list.

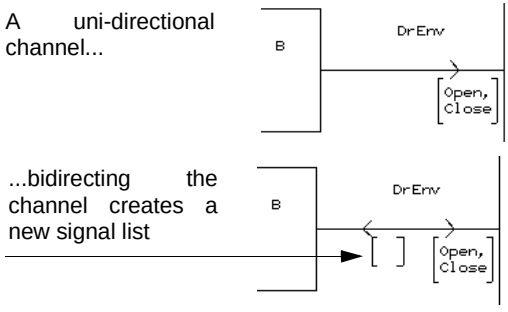


Figure 400: Bidirecting a line

Uni-Directing a Bidirected Line

1. Select the bidirected line.
2. Choose *Unidirect* from the *Edit* menu. A dialog will appear asking what arrow to keep.

Adjusting Lines to the Grid

When you move or reshape interaction lines, the line breakpoints are adjusted to the line grid, which cannot be disabled and has a fixed resolution of 2.5 mm.

Flow line segments and line text attributes are always placed so that they match the resolution of the line grid.

Working with Classes

This section describes how Classes are handled graphically.

The class symbol is a graphical syntax for defining SDL data structures, instead of writing textual newtype definitions. It is allowed in every diagram where textual data definitions are allowed.

When analyzing the SDL specification, class symbols are translated to standard textual newtype definitions, that are used as input to the analyzer.

It is important to notice that, in a diagram, all class symbols with the same class name are treated as if they were a single, merged class. Thus two classes, both named “a_class” but defining two separate operators, would be the same as a single class named “a_class”, defining both operators.

While attributes and operators may be defined and edited in the SDL Editor directly, they may also be edited in the *Browse&Edit Class* dialog, see “Browse & Edit Class Dialog” on page 1943.

The complete definition of all classes may be viewed in the *Browse&Edit Class* dialog, but also, in SDL/PR form, in the *Class information* dialog. The *Class Information* dialog may also be used for traversing among the classes that define the SDL/PR code, see “Class Information” on page 1942.

> **Note:**
>
> Both the *Browse&Edit Class* dialog and the *Class Information* dialog present the aggregate of all graphical class symbols with the same class name.

Class Information

The complete class may be presented in textual (SDL/PR) form. This is done in the read-only *Class information* dialog.

The dialog can be started from the *Windows* menu, and will also be started automatically in response to a *Show symbol* request concerning a class, e.g *Show Error* from the *Organizer Log*. Since the sum of all class symbols with the same name is a single class, some information may be duplicated. The PR-representation of an error is shown in the

class information dialog, and it will be possible to navigate to the symbol(s) that generated the PR-code.

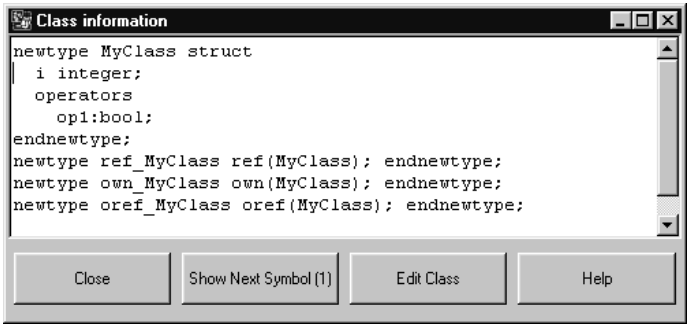


Figure 401 Class Information window

Buttons:

- *Show Next Symbol* shows the next symbol described by the PR code line on which the cursor is placed. The number within brackets shows the total number symbols in the current SDL diagram that that are described by this line. If a line does not refer to any particular symbol, *Show Next Symbol* shows the next symbol belonging to the current class. Each time *Show Next Symbol* is clicked, the next symbol that is described by the current line is shown.
- *Edit Class* will show the *Browse & Edit Class* dialog for the current class.

If the cursor is moved to another line in the text area, the *Show Next Symbol* button is updated.

Browse & Edit Class Dialog

The *Browse & Edit Class* dialog is opened when you select *Class...* from the *Edit* menu. The dialog allows inspection of classes in the SDL diagram.

A class is defined as the union of attributes and operators in the class symbols, and the purpose of the *Browse & Edit Class* dialog is to display this union and ensure consistency when editing this combined information.

The *Browse & Edit Class* dialog is a modal dialog, which is divided into two parts. The browsing functionality is placed at the top part of the dialog, where all classes in the diagram, and all occurrences of each class, are listed in two drop-down menus. Below, in the editing part, the name of the class, its attributes and all its operators are listed.

The *Browse & Edit Class* dialog is available when a single class symbol is selected, unless the class name contains incorrect syntax.

Browse

In the topmost part of the *Browse & Edit Class* dialog, you can browse all class symbols within the current diagram.

Edit

In the lower part of the *Browse & Edit Class* dialog, you can edit the name, attributes and operators of a class. Any changes will propagate into all class symbols of this class within the SDL diagram. The attributes and operators will be presented in a list. This list contains parsed information, derived from the texts in the class symbols. If a particular text contains syntax errors, attributes and operators contained in the same text might be missing from this list.

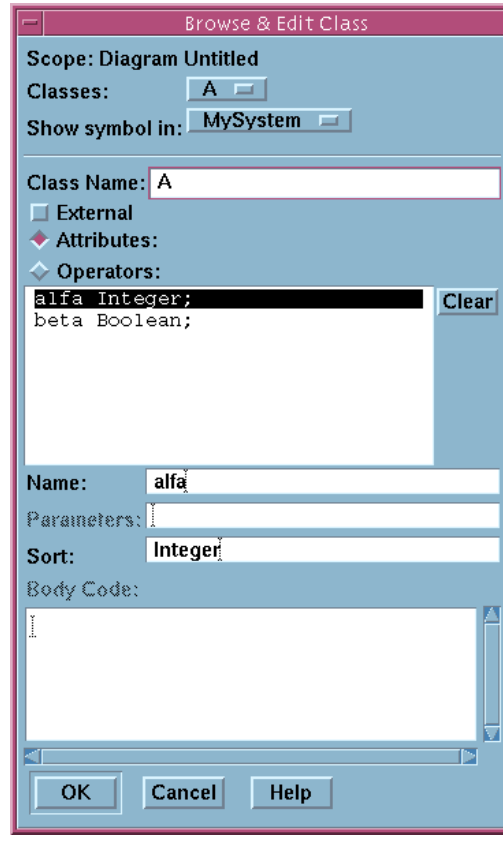


Figure 402: The Browse & Edit Class dialog

The *Scope* Label

The *Scope* label is a non-editable text field, indicating which SDL diagram the *Browse & Edit Class* dialog operates on.

The *Classes* Drop-Down Menu

The *Classes* drop-down menu lists all classes within the current SDL diagram. When a class is selected from this menu, its attributes and operators are displayed in the editing part of the dialog.

The *Show Symbol in* Option Menu

A class can be represented by several class symbols. The *Show Symbol in* drop-down menu lists the name of all pages with occurrences of the current class (the current class is displayed in the *Classes* drop-down menu). By selecting a page name in this menu, the corresponding page will be shown in the work area.

The *Class Name* Field

The *Class Name* field is an editable text field, containing the name of the class being edited.

When the *OK* button is clicked, changes to the *Class Name* field update all occurrences of the class name in all class symbols, within the current SDL diagram.

The *External* Check Box

The *External* check box is checked when a class is externally defined. This means that no PR code will be generated for this class.

When the *OK* button is clicked, changes to the *External* check box update all occurrences of the external flag, in all relevant class symbols using the current class name, within the current SDL diagram.

The *Attributes* and *Operators* Buttons

The *Attributes* and *Operators* buttons are used to select the contents of the attributes/operators list. Clicking Attributes will list all attributes defined for the selected class, while clicking *Operators* will list all operators.

The *Attributes/Operators* List

The *Attributes/Operators* list lists all attributes or all operators defined for the selected class.

The *Clear* Button

The *Clear* button is used to remove the attribute or operator currently selected in the *Attributes/Operators* list.

When the ‘OK’ button is clicked, the selected attribute or operator is removed from all relevant class symbols with the current class name, within the current SDL diagram.

The *Name* Field

The *Name* text field contains the name of the attribute or operator currently selected in the *Attributes/Operators* list, if any.

When the *OK* button is clicked, changes to the definition of the attribute or operator are applied to all relevant class symbols with the current class name, within the current SDL diagram

The *Parameters* Field

The *Parameters* field is an editable text field, containing the parameters of a class operator. It is only editable when an operator is selected in the *Attributes/Operators* list.

When the *OK* button is clicked, changes to the parameter list are applied to the definition of that operator in all relevant class symbols with the current class name, within the current SDL diagram.

The *Sort/Returnsort* Field

The *Sort/Returnsort* field is an editable text field that is editable only when an attribute or an operator is selected in the *Attributes/Operators* list.

Depending on the selection, the field contains:

- the sort of the selected attribute,
- or the return sort of the selected operator.

When the *OK* button is clicked, the sort of the selected attribute, or the return sort of the selected operator, is updated in all relevant class symbols with the current class name, within the current SDL diagram.

The *Body Code* Text

The *Body Code* field is a text field, containing the body code of a class operator. It is only editable when an operator is selected in the *Attributes/Operators* list.

The declaration part of the body code is automatically generated, and all code should be added between the two comments indicating start and

end of the user defined part, respectively. The comments themselves should never be edited. For example, the operator `myOp:Boolean;` will have the following default declaration:

```
operator myOp returns Boolean {
/* Start of user defined body code */
/* End of user defined body code */
}
```

When the *OK* button is clicked, changes to the body code definition are applied to the operator in the class with the current class name, within the current SDL diagram.

The *OK* Button

The *OK* button closes the *Browse & Edit Class* dialog, and updates all appropriate class symbols in the current SDL diagram.

No changes are made in the diagram until the *OK* button is clicked. This makes it possible to specify several changes in the dialog and disregard them later, by clicking the *Cancel* button.

All values are syntactically checked, so it is not possible to add syntax errors to your classes by using the *Browse & Edit Class* dialog. This might make it impossible to delete syntactically incorrect text from class symbols using the *Browse & Edit Class* dialog, since the dialog relies on information obtained by parsing the relevant symbol text compartments. A syntactically incorrect text must then be corrected by editing individual symbols.

Once the *OK* button is clicked, changes cannot be undone.

The *Cancel* Button

The *Cancel* button will close the *Browse & Edit Class* dialog and discard any changes specified in the dialog.

Syntax and Definition of Class Symbols

Name

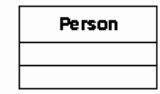


Figure 403 Class symbol

The syntax for the name field is `<SDL_name>`, where `<SDL_name>` is a name in accordance with the SDL lexical rules.

The above class symbol will generate the following newtype definition.

```
newtype Person
endnewtype;
newtype Ref_Person Ref(Person); endnewtype;
newtype Own_Person Own(Person); endnewtype;
newtype Oref_Person Oref(Person); endnewtype;
```

The three pointertypes, Ref_Person, Own_Person and Oref_Person, are generated for use when creating associations and aggregations.

Attributes

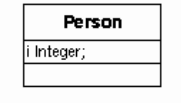


Figure 404 Class symbol with attribute

The syntax for the attribute field is `<name><sort><end>`, where `<name>` and `<sort>` are names in accordance with the SDL lexical rules.

The above class symbol will generate the following newtype definition.

```
newtype Person struct
  i Integer;
endnewtype;
newtype Ref_Person Ref(Person); endnewtype;
newtype Own_Person Own(Person); endnewtype;
newtype Oref_Person Oref(Person); endnewtype;
```

Operators

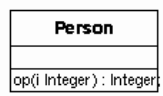


Figure 405 Class symbol with operator

The syntax for the operator field is

```
<name>['('<par>{','<par>*')'][<returns><sort>]<end>
```

where `<par>` is defined as

```
[<kind>]<name>[<colon>]<sort>
```

and `<kind>` is defined as

```
'in'|'in/out'
```

The above class symbol will generate the following newtype definition.

```
newtype Person
  operators
    op: Integer -> Integer;
endnewtype;
newtype Ref_Person Ref(Person); endnewtype;
newtype Own_Person Own(Person); endnewtype;
newtype Oref_Person Oref(Person); endnewtype;
```

Syntax and Definition of Association Lines

Figure 406 Association line

The syntax for the role name of an association is `[<name>][<constraint>]`, where `<constraint>` is in the format `'{'name'}'`.

Associations can be of a number of types, depending on the constraint:

- if there is no constraint, the type is `Ref_<opposite_class_name>`
- if the constraint is “own”, the type is `Own_<opposite_class_name>`

- if the constraint is “oref”, the type is `Oref_<opposite_class_name>`
- if there is a constraint, but not “own” or “oref”, the type is `<constraint_name>`.

The above association will generate the following newtype definition.

```
newtype Company struct
  e Ref_Employee;
endnewtype;
newtype Ref_Company Ref(Company); endnewtype;
newtype Own_Company Own(Company); endnewtype;
newtype Oref_Company Oref(Company); endnewtype;
newtype Employee
endnewtype;
newtype Ref_Employee Ref(Employee); endnewtype;
newtype Own_Employee Own(Employee); endnewtype;
newtype Oref_Employee Oref(Employee); endnewtype;
```

Syntax and Definition of Aggregation Lines

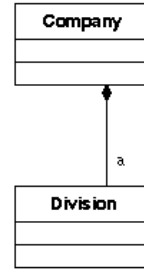


Figure 407 Aggregation line

The above aggregation will generate the following PR code.

```
newtype Company
  a Ref_Division;
endnewtype;
newtype Ref_Company Ref(Company); endnewtype;
newtype Own_Company Own(Company); endnewtype;
newtype Oref_Company Oref(Company); endnewtype;
newtype Division
endnewtype;
newtype Ref_Division Ref(Division); endnewtype;
newtype Own_Division Own(Division); endnewtype;
newtype Oref_Division Oref(Division); endnewtype;
```

Working with Pages

This section describes the more important aspects of page handling.

Each diagram page must have a unique name for identification purposes. This name must be correct in accordance with the naming convention for pages in SDL. Each diagram must contain at least one page, and there is no maximum amount of pages that a diagram may contain.

The *autonumbering* facility available for pages relates to automatically updating the sequential numbering when pages are added or deleted.

When diagrams are opened, the default way of presentation is that the first page of the requested diagram is shown (first in accordance to the order in which pages have been added). This default can be overruled, and you can go directly to a specific page that has been predetermined.

Most of the page managing functions are available through the menu choices on the *Pages* menu.

A number of functions are available through the *Edit* menu choice in this menu:

- *Ordering Pages*
- *Naming Pages*
- *Adding a Page*
- *Renaming a Page*
- *Clearing (Deleting) a Page*
- *Pasting a Page*
- *Transferring to a Page*
- *Printing a Page*
- *Resizing a Page*

Other page functions are also available in the SDL Editor:

- *Applying Autonumbering on Page Names*
- *Removing Autonumbering on Page Names*
- *Designating the Page to Open*
- *Transferring to a Specific Page*
- *Transferring to the Referring Page*
- *Transferring to the Next or the Previous Page*
- *Transferring to the First Page*
- *Transferring to the Last Page*

Ordering Pages

Within an SDL diagram, page locations are set according to the order in which they are added (see "Adding a Page" on page 1954). This order is reflected in:

- The Organizer structure
- The listing order for pages in the *Edit Pages* dialog

To re-order pages, you use the *Cut* and *Paste* button.

Naming Pages

Whenever you name a page, the names that you use must strictly adhere to SDL naming conventions. If you use unacceptable notation (e.g. blank spaces or a semi-colon), you receive a message.

Applying Autonumbering on Page Names

If autonumbering is required, a feature can assign numeric names to the pages in the diagram. The page names will be assigned 1, 2, 3 and so forth.

1. Select *Edit*.
2. Select the page to autonumber in the page list.
3. Toggle the *Autonumbered* toggle button to on in the *Edit Pages* dialog. You are then prompted if you want to autonumber the selected page only or all pages in the dialog which is issued.

   – If you click *Yes*, the SDL Editor assigns the page name the highest assigned numeric value + 1. You are then returned to the *Edit Pages* dialog.

   – If you click *All Pages*, all pages will be assigned numeric names, starting with 1. The numbering will follow the listing order in the page list. You are then returned to the *Edit Pages* dialog.

Removing Autonumbering on Page Names

The autonumbering feature can be turned off. This must for instance be done if you want to assign a specific name to a page.

1. Select *Edit*.
2. In the page list, select the page to remove autonumbering on.
3. Toggle the *Autonumbered* toggle button to off in the *Edit Pages* dialog. You are then prompted to confirm the operation in the dialog which is issued.
   - Clicking *Yes* transfers you to a *Rename Page* dialog where you assign the page a new name. Autonumbering is removed.

Adding a Page

To add a page to an existing diagram:

1. Select *Edit*.
2. In the list of existing pages, select the page to precede or succeed the new page to be added.
3. Click the *Add* button. The *Add Page* dialog is issued.

A faster way to add a page before or after the current page is to select the *Add* menu choice directly. In this case, the *Add Page* dialog is issued directly.

In the *Add Page* dialog:

1. Enter the required page name in keeping with SDL conventions. If the same name is given for two pages they are automatically renamed to name_1 and name_2. The enumartion character, where '_' is the default, used for naming pages with the same name can be set by the preference Editor*Page*EnumerationCharacter. Select *Autonumbered* if you want the pages to be automatically renumbered to incorporate the new page.
2. Select the position of the new page – before or after the current page.
3. Select the type of page required. The dialog box automatically shows the page options that are available to you under SDL rules.

4. Click *OK*. Control is returned to the *Edit Pages* dialog.
5. Click *Done*.

Designating the Page to Open

When a diagram is opened without specifying a particular page, the default is that it is opened at the first page that has been added to the diagram, showing the upper left part of the page.

You can however open a diagram at a specific page in the SDL Editor.

To specify the page to be opened first:

1. Bring up the *Edit Pages* dialog.

   The *Open this page first* field, below the toggle button, reflects the page to be opened first (page 1 in Figure 408).

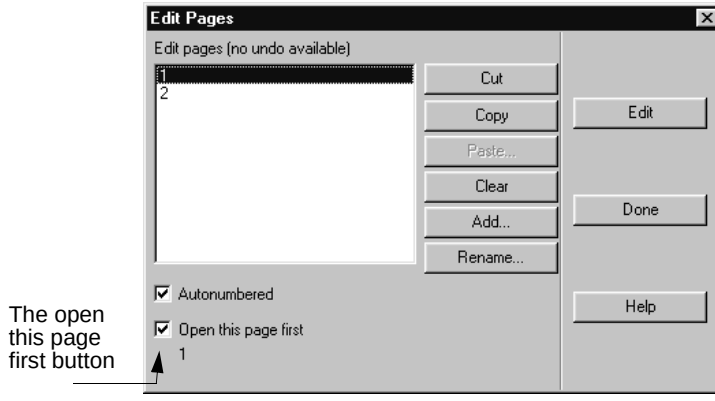


Figure 408: Specifying what page to open first

2. To specify another page, select the page to be opened first in the *Edit pages* dialog.
3. Turn the *Open this page first* button on. The identification of the page to be opened is then placed under the *Open this page first* button.
4. Click *Done*.

Renaming a Page

1. Select *Edit.*
2. Select the page to rename.
   - To rename an auto-numbered page, first click the *Autonumbered* button off. Autonumbering is removed on the selected page and the *Rename Page* dialog is issued. Go to step 4.
3. Click the *Rename* button. The *Rename Page* dialog is issued.
4. Fill in the required new name in keeping with SDL naming conventions. For block and process diagrams, an option is available to also change the page type.
5. Click *OK*. Control is returned to the *Edit Pages* dialog.
6. Click *Done*.

Clearing (Deleting) a Page

If there is only page in a diagram, you cannot remove it. A message is then shown.

Caution!

Clear pages with caution as there is no *Undo* option available.

1. Select *Edit*.
2. Select the page to be deleted in the page list.
3. Click on the *Clear* option and then on *Clear* to delete the page.
4. Click *Done*.

Pasting a Page

1. Open the SDL diagram where to paste the page.
2. Select the *Edit* menu choice.
3. In the list of existing pages, select the page to precede or succeed the new page to be pasted. Click the *Paste* button. The *Paste Page* dialog is issued.

4. Enter the required page name in keeping with SDL conventions (the name must be unique within the diagram). Otherwise, select *Autonumbered* if you want the pages to be automatically renumbered to incorporate the page to be pasted.
5. Select where to paste the page - before or after the current page.
6. Click *Paste*. Control is returned to the *Edit Pages* dialog.

   If there are reference symbols on the pasted page all the underlying substructure is also pasted. See "Cutting, Copying and Pasting Reference Symbols" on page 1917.
7. Click *Done*.

Transferring to a Page

There are several ways to transfer to another page of the current diagram. When you transfer to a page, it will be placed on the top of the stack in the window if you are using one window only. Otherwise, that page will be displayed in a window of its own.

Transferring to a Specific Page

- Transfer to a page, located not further than two pages from the current page, by selecting that page in the *Pages* menu.
- In the *Edit Pages* dialog, transfer to a page by clicking on the required page and then click on the *Edit* button.
  - Alternatively, double-click the page icon in the Organizer.

Transferring to the Referring Page

To transfer to the page where the current page is referenced from (i.e. where the SDL reference symbol referring to the current diagram is located):

- Select *Edit Reference Page* from the *Pages* menu or the *Reference Page* quick-button.

Transferring to the Next or the Previous Page

You may transfer to the next or previous page in the diagram. The order is specified according to the listing order in the *Edit Pages* dialog.

To transfer to the next or previous page:

- Use the quick-buttons for *Previous Page* or *Next Page*.

Transferring to the First Page

Select *First* from the *Pages* menu. You are transferred to the first page of the diagram.

Transferring to the Last Page

Select *Last* from the *Pages* menu. You are transferred to the last page of the diagram.

Printing a Page

The SDL Suite allows to print individual pages from the SDL Editor. You can also restrict the scope of printing to a part of the page. A number of options which affect the resulting printout are possible to specify.

Resizing a Page

1. Transfer to the page to resize.
2. Select *Drawing Size* from the *Edit* menu. A dialog is issued, showing the current width and height.
3. Type in the new values. Click *OK*.

> **Note:**
>
> If you enlarge a page so that the page size becomes wider or larger than the physical page size defined in the *Print Options*, your printouts will require more pages than before, if you use a fixed printout scale.

Working with Windows

Each SDL Editor window shows one SDL page. The SDL Editor is a multi-window tool, allowing you to open new windows on a page and close windows that are no longer needed.

A newly opened window is a mirror image of the window from which the selection to open a window was made. Multiple windows may be opened (one at a time) and they are all identical with the first one. Any subsequent modifications made will reflect the same in either the original window, or any of the newly opened window(s). This option affords the possibility of viewing in more than one window, and can ideally be used in conjunction with the scaling factor when looking at a detailed page.

For information on how you open:

- The Grammar Help Window, see "Opening the Grammar Help Window" on page 1972.
- The Signal Dictionary Window, see "Opening the Signal Dictionary Window" on page 1984.

Opening and Closing Windows

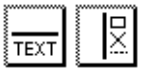
Standardized functions for opening and closing windows are provided in the *Window* menu: *New Window* and *Close Window*. If you close the last window opened by the SDL Editor, the editor will exit.

Hiding and Showing Parts of the SDL Editor Window

You can show and hide the various component parts of the SDL Editor window. **On UNIX**, if they are hidden, it increases the drawing area available to be used in the creation or modification of pages of a diagram.

You find all the available options for hiding and showing parts of the window in the dialog box accessed via the *Window Options* command on the *View* menu. You can set all of these options as preferences.

All of these options are turned on and off with a toggle button. Some of these options are available with quick-buttons:



- The quick-button for *text window* on / off or *symbol menu* on / off.

  – If you click *All Windows* it applies the values to all instances of the SDL Editor windows.

Editing Text

General

There are several ways of editing text in the SDL Editor, i.e.

- In the drawing area
- In the text window
- In an external text editor

Two support functions are provided, namely:

- Grammar Help, see "Using Grammar Help" on page 1972.
- Signal Dictionary, see "Using the Signal Dictionary" on page 1983.

Editing in the Drawing Area and Text Window

The SDL Editor allows you to edit textual objects directly in the drawing area.

If you select a text attribute to an object in the SDL Editor window, a text cursor is inserted directly in the text associated with the object, and the *text window* will be updated to contain the text. This is explained in "Selecting a Symbol That Has Associated Objects" on page 1903 and "Selecting a Line That Has Associated Objects" on page 1932.

When you type text in the drawing area, all accelerator keys are interpreted as input to the text. As an example, the `<Delete>` key will in text editing mode delete a character, but in non-text editing mode it will remove (*Clear*) an entire symbol. This means that you can only use text editing keyboard accelerators like `<Home>`, `<End>`, `<Ctrl+A>` and `<Ctrl+E>` in text editing mode.

When editing text directly in the drawing area it is possible to select text by dragging the mouse over the text or select words by double-clicking.

You can extend or reduced the selection by clicking while pressing the <shift> key.

To edit large texts it might be more convenient to use the *Text Window* to edit the text. If you select more than one text object, the text window is not updated. Each line (except for the last line) in the text window is terminated by a carriage return, and may consist of any number of legal characters.

Regardless of whether you edit directly in the drawing area or in the text window, the SDL Editor makes sure that the contents of both displayed texts are consistent.

A context sensitive syntax check is performed on all texts being edited. If a syntax error is found a red bar underlining the text will appear where the first error occurs in the text. See also "Syntax Rules when Editing" on page 1852.

You can also edit text in an external text editor by using the *Connect to Text Editor* command in the *Tools* menu. This allows you to use the Text Editor, the Emacs text editor **(on UNIX)** or MS Word **(in Windows)** to edit large pieces of text.

The SDL Editor also supports editing text outside its context, by transferring text to / from a file.

Text Window

On UNIX, the text window is a pane of the SDL Editor window.

In Windows, the text window is a floating, resizeable and moveable window that can be placed anywhere on the screen. A single text window is shared by all instances of the SDL Editor currently running.

The text window contains two menus described in:

- "The File Menu of the Text Window" on page 2029
- "Edit Menu" on page 2004

Hiding and Showing the Text Window

You can hide or show the text window with the *Window Options* command from the *View* menu, or by using a quick-button.

In Windows: When visible, the text window will always be placed on top of the SDL Editor window.

Basic Text Editing Functions

Delayed Updating of the Drawing Area

When you edit text in the text window, both the text window and corresponding text in the drawing area will be updated. The the drawing area will be updated after a slight time delay and this can be set by an editor preference.

If you try to type an illegal character, you will be warned by a beep.

Standardized Text Editing Functions

When you edit text, you can:

- Position the cursor by clicking or by using the arrow keys.
- Insert characters after the current position of the text pointer.
- Delete one character backward by pressing `<backspace>`.
- Delete one character forward by pressing `<delete>`.
- Replace selected text by typing.
- Delete selected text by pressing `<backspace>` or `<delete>`.
- Select text by dragging or by clicking and `<Shift>`-clicking.
- Select a word by double-clicking it.
- Select a line by triple-clicking it.

Searching and Replacing Text

In the SDL editor, you can search for and replace text in a page or in a diagram. In the Organizer, you can search for and replace text in a diagram or in a diagram structure that is managed by the Organizer. In both the SDL editor and the Organizer, there are two search variants:

- Dialog search
- Fast search

Searching for Text in a Diagram using Dialog Search

To search for text in the current page from the SDL editor:

1. Select *Search* from the SDL editor's *Tools* menu or click the SDL editor quick-button for *Search*. The *Search dialog* is issued.

2. Specify the text to be found in the *Search for* text field.
   - Click the *Consider case* button on to search case sensitive.
   - Click the *Wildcard search* button on to use wildcard matching, where an asterisk ('*') matches a sequence of zero or more characters of any kind.
   - Turn the *Search all pages* button on to specify that the scope of search should comprise all pages in the diagram. If you turn the button off, the scope of search will only comprise the current page.
   - Click the *Whole word search* button on to search only whole words.
3. Click the *Search* button. The object where the next occurrence of text is found will be selected and the text window updated accordingly.
   - The diagram that the tool is currently processing is displayed to the right of the *Search In* field.
4. Click *Close*.

Searching and Replacing Text using Dialog Search

1. Open the Search dialog as described in "Searching for Text in a Diagram using Dialog Search" on page 1962.
2. Specify the text to be found in the *Search for* text field.
3. Specify the text to be replaced with in the *Replace with* text field.
4. Click the *Search* button to find the first occurrence.

5. If the text is found, the object where the next occurrence of text is found will be selected and the text window updated accordingly. You may use any of the following options:
   - Search further by clicking *Search* repeatedly
   - Replace the text by clicking *Replace*
   - Replace and continue search with the *Replace & Search* button.
   - Replace all occurrences with the *Replace All* button.
6. Close the Search dialog by clicking *Close*.

Fast Search

Fast search is like dialog search but without the dialog. The setting from the last dialog search is used for *Wildcard search*. Fast search do never consider case, and does always search all objects and all pages. To invoke Fast search, type ctrl+F when the SDL editor is not in text editing mode. If you are in text editing mode, you can click in the diagram background first to get out of text editing mode.

When Fast search is invoked, the message area displays the text that will be searched for. Initially, the text is *Search for:*. When you type on the keyboard with the mouse pointer over the drawing area, the characters will turn up in the message area. When you have typed the text pattern to search for, press enter or return to start the search operation.

The same search operation as for Dialog search is used. If the search operation finds a match, you can search for another match by pressing enter or return once more.

To finish the Fast search operation, click in the drawing area or select another operation.

The next time Fast search is invoked with ctrl+F, the search text that was used the last time is proposed as a search text once more. To use it, press enter or return. To use another search text, press ctrl+F once more or the delete key several times, to erase the search text.

Editing Text by Using an External Text Editor

Whenever a text object is selected and the text is visible in the text window, the command *Connect to Text Editor* in the *Tools* menu is avail-

able. This command opens an external text editor containing the text of the selected object. Which external text editor to start is defined by the preference SDT*TextEditor, which can be set either to the Text Editor or to the Emacs text editor **(on UNIX)**.

From this point on, you can only edit the text by using the external text editor.

The text in the SDL Editor is updated every time the external text editor saves the text. When the text is no longer edited in the external text editor, the editing control returns to the SDL Editor.

Importing / Exporting Text

The SDL Suite allows you to import and export text from / to a file. You can, for instance, import text from files that contain SDL/PR into text symbols and export the contents of text symbols to files.

Importing Text from a File

Caution!

Importing text from a file replaces existing text as well. Use the *Undo* facility to revert to the original text, as long as the symbol is selected – deselecting the symbol will disable *Undo*.

You can import the contents of a text file into an object managed by the SDL Editor in the following way:

1. Select the object where you want to import the contents of a file.
2. Select *Import* from the text window's *File* menu.
3. Specify the file to import.
4. Click *OK*. This **replaces** the contents of the text window with the contents of the file you have specified in the file selection dialog.

Exporting Text from a File

To export the contents of the text window into a file:

1. Select the object which text you want to export to a file.
2. Select *Export* from the text window's *File* menu.

3. Specify the file to export the text to.
4. Clicking *OK* possibly creates a new file and replaces the contents of the file with the contents of the text window.

Copying, Cutting and Pasting Text

In the text window, you can cut, copy and paste text between different symbols, lines and text attributes. You can also transfer text between different applications.

Programmable Function Keys (UNIX only)

On UNIX, the SDL Suite allows to tie a function key to a defined text string. When you type that defined function key, the programmed text string will be inserted at the current cursor location. You can customize your own programming of function keys.

Global SDL Suite Resources

The function keys are set up as X resources. It is possible to set up both system default and user-defined X resources, allowing you to customize your environment. The X resources are defined in a file that is common for all SDL Suite users, namely

```
/usr/lib/X11/app-defaults/SDT
```

To program the function keys, insert following lines anywhere into the `SDT` file:

```
/* Any suitable comment */
SDT*XmText.translations: #override \n\
<Key>F1: insert-string("F1Text") \n\
<Key>F2: insert-string("F2Text") \n\
<Key>F3: insert-string("F3Text") \n\
<Key>F4: insert-string("F4Text") \n\
<Key>F5: insert-string("F5Text") \n\
<Key>F6: insert-string("F6Text") \n\
<Key>F7: insert-string("F7Text") \n\
<Key>F8: insert-string("F8Text") \n\
<Key>F9: insert-string("F9Text")
/* Note the absence of \n\ on line 9 */
```

Note:

Omitting to define some of the function keys is permissible.

User-Defined Resources

You can define your own function keys. You do this by defining the X resources described above in a personal copy of the definition file and to store that file it into your home directory:

```
~username/SDT
```

Alternatively, any directory designated by the environment variable `XAPPLRESDIR` can be used.

Restrictions

- You can only define one line for each function key. If you attempt to define more than one line into one function key may cause an unpredictable result when you press that key.

  For instance, it is not certain that the following line will produce the expected result:

  ```
  <Key>F1: insert-string("F1Line1\nLine2") \n\
  ```

- You can define the keys F1–F9.

Changing Fonts on Text Objects

You may change the font faces and font sizes used in the textual objects displayed by the SDL Editor. All textual objects use the same font faces and font sizes, meaning that you can neither change them individually nor change them during an SDL Editor session.

There is one exception though – it is possible to use a separate font for text symbols. This is useful in process diagrams where you may want a proportional font in most symbols to save space, while you may want a non-proportional font in text symbols to be able to align words on different lines.

The font faces available depend on the target system on which you are running the SDL Suite.

Defining What Font to Use

To modify the desired font size and font face, you must use the Preference Manager. See chapter 4, *Managing Preferences*.

Textual Objects Preferences

When the setting is in effect, the SDL Suite will use the font face names given by the preference settings

Editor*FontText*ScreenFontFamily
Editor*FontText*PrintFontFamily

to select font face names. Note that in this way you can select different font names for screen and for print.

On UNIX, if you leave the Editor*FontText*ScreenFontFamily preference setting empty, you will edit your documents using the *SDT Draft* font, but print them using the font you specified with the Editor*FontText*PrintFontFamily setting.

Text Symbol Preference

For text symbols, the preference Editor*FontText*TextSymbolFontFamily is used both for on-screen viewing and printing.

Supported Font Faces

On UNIX, the availability of font faces is determined by the version of the X Windows server which is running. With a revision 5 or higher (X11 R5), scalable fonts are supported. In that case, the available list of predefined font faces would be:

- Times
- Helvetica
- Courier
- SDT Draft (mapped to the Schumacher font)
- Other (mapped to a user-defined font)

In Windows, the availability of font faces is determined by the TrueType fonts that are currently installed on the computer (use for instance the Windows Control Panel to determine what is available).

Default Font Face

The default font face is Helvetica. **On UNIX**, if scalable fonts are not supported, the font face will be replaced by a *Schumacher* font (which can be used in all circumstances).

Default Font Sizes

The default font sizes are as follows:

Page Font Sizes (except overview pages)	
Text Object:	**Font Size**
Kernel heading, texts in reference symbols, channel names, signal route names, gate names	12 points
Other text objects	9 points

Overview Page Font Sizes	
Text Object:	**Font Size:**
All texts except signal lists	10 points
Signal list symbol	8 points

Setting Alignment

Normally the alignment of the text inside symbols are automatically set. However, you can use the preference setting Editor*Font-Text*TaskSymbolLeftAligned to set the alignment for the task symbol, procedure call symbol, macro call symbol, create request symbol and the save symbol. The default value is “off”, meaning that the text will be centered, but if you set it to “on” the text in these symbols will be left aligned.

Determining Which Scalable Fonts Your Server Can Access (UNIX only)

On UNIX, use the `xlsfonts` command to list installed fonts. Font names containing 0 for width and height are scalable.

How to Determine what Fonts are Available

From the OS prompt, typing:

```
hostname% xlsfonts | grep "\-0\-0\-" | more
```

will return a list of accessible scalable fonts.

Scalable Fonts Under R5 Servers

To use scalable fonts under X11R5 you must normally first connect to a font server.

How to Start the Font Server

1. Start the font server on any local host:

   ```
   hostname% fs
   ```

2. Connect the server to `fs` indicating which host the font server is running on (which can be the same host that the X server is running on):

   ```
   hostname2% xset +fp tcp/<hostname>:7000
   ```

For further information see the X11R5 documentation or use `man fs` to read the manual page describing the font server you are running.

Disabling Font Scaling (UNIX only)

On UNIX, if the fonts look poor on the screen, a possible work-around is to disable the scaling option.

> **Note:**
> Disabling font scaling effectively disables WYSIWYG!

To do this, you should edit the `SDT` resource file.

1. Open the file `SDT` in a text editor.
2. Locate the line with the text: `SDT*sdtUseScalableFonts`
3. Change the line to `SDT*sdtUseScalableFonts: false`
4. Save the file and restart the SDL Suite environment.

Grammar Help and Signal Dictionary

General

Besides the context sensitive syntax check performed on the texts, there is a *grammar help* support function. It assists you in entering the correct text, according to the SDL grammar, in the selected text attribute to an object.

Designing using SDL implies to large extent defining, sending and receiving signals. A *signal dictionary* assists you in reducing the time it takes to find out names and parameters for a signal that you already have used somewhere in a diagram. The signal dictionary also incorporates timers.

Keyboard Accelerators (UNIX only)

In addition to the standard keyboard accelerators, described in "Keyboard Accelerators" on page 35 in chapter 1, *User Interface and Basic Operations*, the Grammar Help and Signal Dictionary window features the following:

Accelerator	Command / functionality
`g`	Select the *Grammar* section if the option is enabled and bring the separator into view
`u`	Similar to `g` but applies on *Up*.
`t`	Similar to `g` but applies on *This*.
`d`	Similar to `g` but applies on *Down*.
`a`	Similar to `g` but applies on *All*.
`m`	Similar to `g` but applies on *MSC*.
`e`	Similar to `g` but applies on *External*.
`Ctrl+g`	Toggles the *Grammar* option on / off (see "Grammar" on page 2031). When the option is enabled, brings the Grammar separator into view and selects it.
`Ctrl+u`	Same as `Ctrl+g`, but applies on the *Up* option.

Accelerator	Command / functionality
Ctrl+t	Same as Ctrl+g, but applies on the *This* option).
Ctrl+d	Same as Ctrl+g, but applies on the *Down* option.
Ctrl+a	Same as Ctrl+g, but applies on the *All* option.
Ctrl+m	Same as Ctrl+g, but applies on the *MSC* option.
Ctrl+e	Same as Ctrl+g, but applies on the *External* option.
Ctrl+f	Finds the **last** occurrence of the first word in the currently selected object in the SDL Editor drawing area and selects it in the *This* section. The first word is defined as the text string from the start of the object's text and until one of the following characters: ' ' (space) ',' (comma) '(' (left parenthesis) <TAB> '\n' (newline)

Using Grammar Help

The Grammar Help window is a multi-functionality window; it can also provide signal dictionary capabilities. These functions are further described in "Using the Signal Dictionary" on page 1983. What functionality is provided depends on the options defined in the *Select* menu.

Each SDL Editor window has its associated Grammar Help window.

Opening the Grammar Help Window

- Select the *Grammar Help* menu choice from the *Windows* menu.

  The *grammar help window* is issued (see Figure 409 on page 1974).

Requesting Grammar Help

If you select the object you need assistance on, you will see the keywords and options available for use in given situations, and the corresponding reference to sections of the ITU Z.100 SDL Definition, followed by the grammar syntax of the meta language.

To request grammar help:

- Select the object of interest. The grammar help window contents are automatically updated to list the following:
  - The nature of the selected object
  - The grammar for that object
  - The various “use cases”.

Inserting Text

This operation inserts the contents of the grammar help window into the SDL Editor text window.

To insert the text related to a given “use case”:

1. Locate the “use case” of interest. The left part of the window provides a list of situations, named using some abbreviation that associates to the situation.
   - The first item(s) in the list reads *GRAMMAR*, possibly with a suffix that informs you about the grammar context. You can take advantage of the GRAMMAR items either as using the references to Z.100 if you need more detailed information about the selected sort of object, or if you need to read more about the concrete grammar.

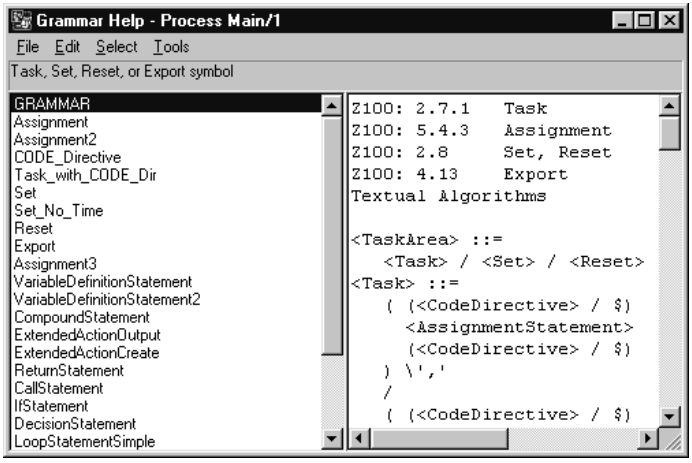


Figure 409: The grammar for a task symbol

– The next items contains a number of “use cases”. As an example, if you select a task symbol, the first “use case” reads *Assignment*, while the second reads *Assignment2*. This is to be interpreted as:
An assignment statement,
Multiple assignment statements in a task symbol.

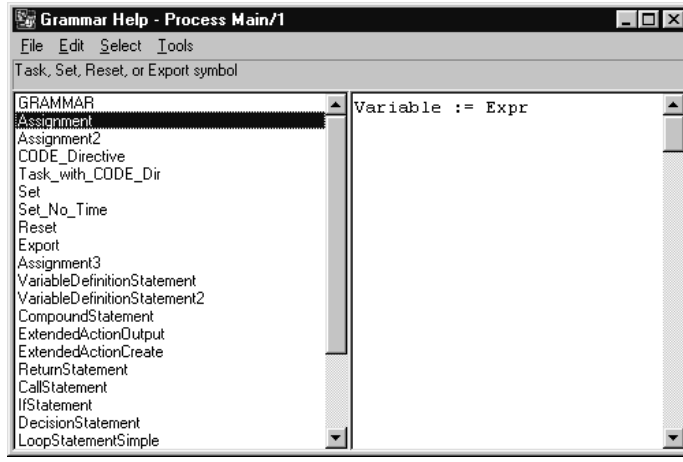


Figure 410: The Assignment "use case"

2. Click on the item of interest to select it.
3. Select *Insert* from the *Edit* menu or double-click the item in the list. The contents of the right part of the window are inserted into the SDL Editor text window, at the current I-beam cursor position.
4. Finally, replace the formal notation with the appropriate values, variables, signals, etc. used in your diagram.

Replacing Text

This operation signifies replacing the contents of the SDL Editor text window with the contents of the grammar help window.

- Proceed as described in "Inserting Text" on page 1973, but select *Replace* from the *Edit* menu.

Customizing the Grammar Help

With the SDL Suite, a standard grammar help template file is provided. In addition to this, the SDL Editor allows you to use your own templates and to merge these with the standard templates.

To create grammar help definitions:

1. Copy an existing grammar help file.
2. Edit the file using any text editor (the grammar help file is an ASCII file). See "The Grammar Help File" on page 1978 for a description of the contents and syntax of the file.

To use another grammar template definitions:

- Select *Load* from the *File* menu. In a File Selection Dialog you then specify what template definitions to load.

To merge the current grammar template definitions with another one:

- Select *Load* from the *File* menu. In a File Selection Dialog, you then specify what template definitions to merge with the current.

The *Symbol* Label

The *symbol label* field is a non-editable text field that displays the type of symbol that is currently selected in the SDL Editor.

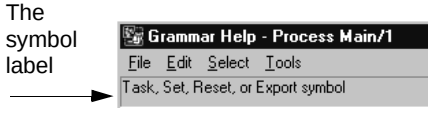


Figure 411: The symbol label

The symbol label reads "No single symbol selected" in the following circumstances:

- No symbol or line is currently selected in the SDL Editor
- More than one symbol is selected
- The symbol type is not defined in the *Grammar Help* file.

The *Name* Field

The *name field* consists of a scrollable list that contains a list of names of templates associated with the currently selected symbol.

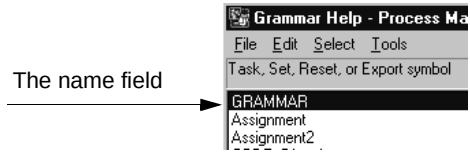


Figure 412: The name field

The list is updated automatically each time an object in the SDL Editor's drawing area is selected, to reflect the templates that are currently available for that object. By default, the first item in the list is selected.

The name field is empty in the following circumstances:

- The symbol label reads "No single symbol selected"
- The symbol type information is missing in the *Grammar Help* file.

When selecting an item in the list, the corresponding template definition appears in the grammar field, allowing to check its contents before inserting it into the text window (see "Insert" on page 2030 and "Replace" on page 2030).

The *Grammar* Field

The *grammar field* is a scrollable field from which text can be copied. Its contents are updated to reflect the definition for the currently selected item in the name field.

The grammar field is empty if no item is selected in the name field (or if the name field is empty).

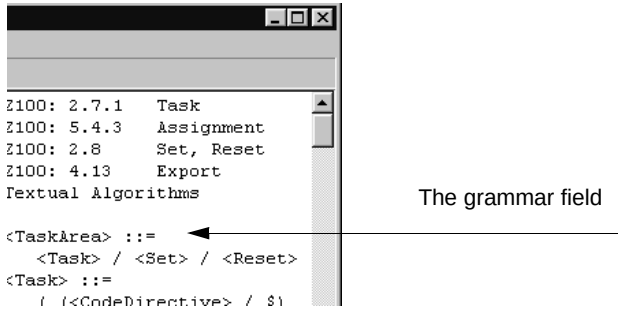


Figure 413: The grammar field

To copy and paste text from the template definition field to the text window, you can:

- Use the *Insert* command, described in "Insert" on page 2030.
- Use the *Replace* command, described in "Replace" on page 2030.
- Copy and paste by using the graphical environment clipboard functions.

The *Grammar Help File*

This section is a reference for the format of the files that contain SDL Grammar Help definitions.

The template definition file format is line-oriented, and uses two separator characters chosen so as to not interfere with the possible contents of templates appearing in the file.

> **Note:**
>
> The characters `'@'` and `'$'` are separator characters. They are **reserved for that purpose.**

The template definition file is divided into **sections** defining templates for a particular symbol type. A section begins with a *Grammar Help Declaration* part and ends with a *Grammar Help Definition* part.

The **grammar help declaration** of a new section has the following syntax:

```
@<editortype>@<symboltype>[@[<symbolcomment>]]
```

<editortype>, <symboltype> and <symbolcomment> are to be replaced with the adequate text strings.

Comments in Grammar Help files are also permissible.

Example 322: The Declaration of a Channel Template ––––––––––––

```
@SDL@CHANNEL@Channel symbol
```

The declaration is to be interpreted as:

- `SDL`
  Valid for the SDL Editor only
- `CHANNEL`
  A channel line
- `Channel Symbol`
  An informative text that will be displayed to you (see "The Symbol Label" on page 1976).

––

editortype

editortype may currently be of one of the keywords:

- - The section contains templates relevant for the SDL Editor.
- - The section contains templates for the Message Sequence Chart Editor.

> **Note:**
> Grammar Help is currently not supported by the MSC Editor. The MSC keyword is reserved for future functionality extensions.

symboltype

symboltype is defined per editor and the allowed names are the following:

```
ADDITIONAL_HEADING_BLOCK
ADDITIONAL_HEADING_BLOCK_TYPE
ADDITIONAL_HEADING_PROCEDURE
ADDITIONAL_HEADING_PROCESS
ADDITIONAL_HEADING_PROCESS_TYPE
ADDITIONAL_HEADING_P_AS_SERVICE
ADDITIONAL_HEADING_SERVICE
ADDITIONAL_HEADING_SERVICE_TYPE
ADDITIONAL_HEADING_SYSTEM
ADDITIONAL_HEADING_SYSTEM_TYPE
BLOCKSUBSTRUCTURE
BLOCK_REF
BLOCK_TYPE_REF
CALL
CHANNEL
CHANNELSUBSTRUCTURE
CONNECTION_POINT
CONNECTOR
CONTINUOUS_SIGNAL
CREATE
DECISION
ENABLING_CONDITION
GATE
INPUT
KERNEL_HEADING
KERNEL_HEADING_BLOCK
KERNEL_HEADING_BLOCK_TYPE
KERNEL_HEADING_PROCEDURE
KERNEL_HEADING_PROCESS
KERNEL_HEADING_PROCESS_TYPE
KERNEL_HEADING_SERVICE
KERNEL_HEADING_SERVICE_TYPE
KERNEL_HEADING_SYSTEM
KERNEL_HEADING_SYSTEM_TYPE
MACROCALL
OPERATOR_REF
OUTPUT
CLASS
PACKAGE_REF
PAGENUMBER
PRIORITY_INPUT
PRIORITY_OUTPUT
PROCEDURE_REF
PROCEDURE_START
PROCESS_REF
PROCESS_TYPE_REF
RETURN
SAVE
SERVICE_REF
SERVICE_TYPE_REF
```

```
SIGNALROUTE
START
STATE
STOP
TASK
TEXTSYMBOL_BLOCK_SUBSTR
TEXTSYMBOL_MACRO
TEXTSYMBOL_PROCEDURE
TEXTSYMBOL_PROCESS
TEXTSYMBOL_P_AS_SERVICE
TEXTSYMBOL_SERVICE
TEXTSYMBOL_SYSTEM
TRANSITION_OPTION
```

Note:

If the same section appears more than once, all template definitions under those sections will be available.

symbolcomment

symbolcomment is an optional feature, which allows specification of a text to be associated with each *symboltype*. The text for a specific *symboltype* appears in the symbol type field of the Grammar Help window whenever a symbol of that type is selected in the editor.

Comments

Also, comments may appear in the file using the following syntax:

```
@COMMENT@<commenttext>
```

Note:

Such a comment signals the end of the previous section and should only be used before another section.

Grammar Help Definition

A SDL grammar template has significance only if it is located after a valid declaration. The template will be added to the list of templates for that section.

A template definition is started with a line beginning with a '$' sign and continues until either:

- A line beginning with a '$' sign is read.
- A line beginning with a '@' sign is read.
- End of file is encountered.

The **syntax** of a template definition is simply:

```
$<templatename>'Newline'
<multiple lines constituting the template
definition>
```

Example 323: The definition of a channel example ––––––––––––––––

```
$GRAMMAR
Z100: 2.5.1 (p.45)
Z100: 2.5.5 (p.50) Signal list

<ChannelName> ::= <Name>
<SignalList> ::=
  ( <SignalName> / '(' <SignalListName> ')' )",'
$SignalList
SignalName, SignalName
$SignalList2
SignalName, SignalName, (SignalListName)
```

––

Note:

Neither leading nor trailing newlines will be added to the template definition. These can be added to the template simply by adding a leading empty line and/or a trailing empty line in the template definition.

File Handling

When a Grammar Help window is opened, the SDL Editor will try to locate a SDL grammar help file and to load it, if found. The search order is as follows:

1. The SDL Editor starts by fetching the preference value `Editor*TemplateFile`.
2. If no value is specified for the preference parameter, it will use the **default file name** `sdt.tpl`.
3. The directories where the SDL Editor will search for the file are:
   - The current directory
   - Your home directory (`$HOME` **on UNIX** and `%HOME%` **in Windows**)
   - The installation directory (`$sdtrelease` **on UNIX**)
4. If no file can be located, either with a specified name or the default name, you get the option of searching manually (i.e. from a File Selection Dialog) for files with the standard file name extension `.tpl`.

5. If you decline by clicking *Cancel*, the Grammar Help window will appear without contents. You can now locate a grammar help file by choosing *Load* (or *Merge*, which will have the same effect in this case).

Using the Signal Dictionary

Messages from a Message Sequence Chart and signals from an external signal dictionary may be included into the SDL Editor's signal dictionary.

All functionality is provided in the *signal dictionary window*. Each SDL Editor window has its associated Signal Dictionary window.

> **Note:**
>
> The signal dictionary function requires a file with grammar help templates (`sdt.tpl`) in order to function properly. See "The Grammar Help File" on page 1978.

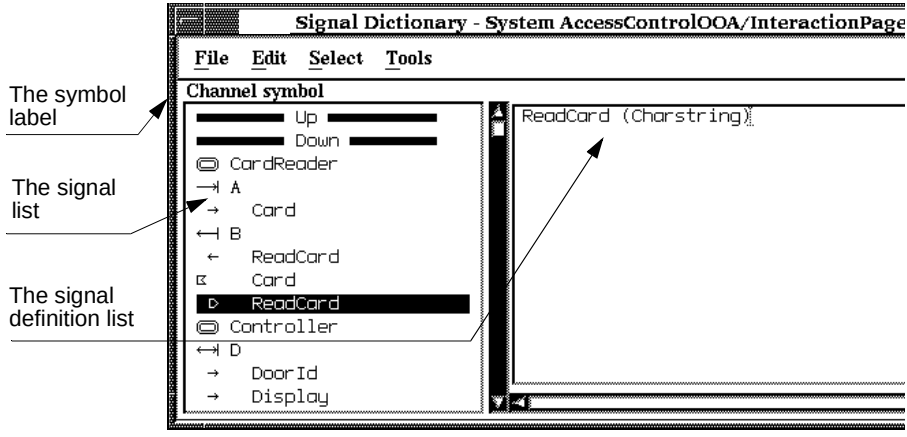


Figure 414: The Signal Dictionary window **(on UNIX)**

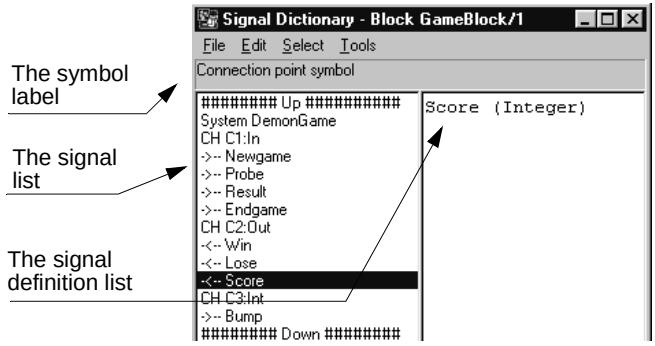


Figure 415: The Signal Dictionary window **(in Windows)**

Signal Dictionary Update

The signal dictionary function stores information about signals in an *information server*. The information server updates its contents **each time an SDL diagram is saved**. When the diagram is saved by the SDL Editor, the SDL Analyzer is invoked and produces a mirrored image which is an SDL/PR description of the SDL/GR diagram. That information is then loaded into the information server which serves the SDL Editor with information upon request. All of this is done automatically.

> **Note:**
>
> In order to extract signal information from a diagram, the SDL Suite requires the diagram to be syntactically correct in the sense that SDL/PR code can be generated from the SDL/GR diagram without encountering errors.

Opening the Signal Dictionary Window

- Select *Signal Dictionary* from the SDL Editor *Windows* menu. The signal dictionary window is issued (see Figure 414 on page 1983 and Figure 415 on page 1984).

Setting the Mode to Graphical or Textual (UNIX only)

On UNIX, the signal dictionary window can take advantage of your computer's font capabilities to present the information in a graphical way, using SDL-like notation. This is the default mode.

If the information is presented using strange characters, your terminal does not support the font family used by the signal dictionary window. You should use the textual mode.

To turn the mode to textual:

1. Set the environment variable `SDLENOGRAPHICS`
   - If you are to run several Signal Dictionary sessions in this mode, you may want to set the environment variable in a file that you will automatically source (e.g. `.cshrc`)
2. Exit all SDL Suite tools.
3. Restart the SDL Suite to have the environment variable affect the behavior.

Requesting Signal Dictionary Services

When you select an object where a reference or definition of a signal makes sense, the signal dictionary window is automatically updated to reflect the signals and *signal conveyors*[1] that are available according to what options you have selected.

The signal dictionary function responds when selecting the following objects (see "Object" on page 1995 for more details):

- channel
- signal route
- connection point
- gate
- signal input
- signal output
- priority input
- save of signal
- text symbol

Updating the Signal Dictionary

The signal dictionary is automatically updated each time you save an SDL diagram or an MSC that is referred to in the Organizer's SDL System Structure chapter.

1. The term *signal conveyor* denotes a communication path that conveys signals; a channel, signal route, a gate or connection point.

When modifying an SDL diagram, you may have unintentionally introduced SDL constructs that are incomplete and thus cause the signal dictionary to fail in providing signal information for that diagram.

Where the signal dictionary fails on a diagram, a bug symbol **(on UNIX)**, or an error message **(in Windows)** will appear after the diagram name in the signal list. Correct the source of error and save the diagram again. For a reference on signal dictionary errors, see "Error Notification" on page 2001.

Specifying the Signal Dictionary Options

You can customize the signal dictionary to fit your needs, by restricting or extending the amount of information that is presented through a set of options that you can activate or deactivate.

Depending on the method you are following when designing with SDL, you should activate the required options.

Next follows a guide for when to use the available options.

To enable an option:

1. Activate the *Select* menu.
2. Toggle the appropriate option on by selecting it (an option which is enabled is indicated with an asterisk preceding the option name). Selecting the option once again disables it.
   - Alternatively, select the *Options* menu choice and toggle the buttons on or off. When satisfied, click *OK*.

> **Note:**
>
> The more options are enabled, the more information is computed by the SDL Editor and the more time it takes.

Bellow follows a few general guidelines for enabling the various options, depending on the approach you are following when designing with SDL.

Working Top-Down

This expression means starting by designing the root diagram (e.g. the system diagram) and continuing with the block diagrams, next the process diagrams and so forth.

If you follow this approach, selecting the *Up* option will enable access to signals used one level up in the SDL hierarchy. This option is enabled by default.

– You may also type `<Ctrl+u>` to toggle the option. Typing **`u`** brings the *Up* option into view (if enabled).

Working Bottom-Up

Following this method means starting with the diagrams at a deeper level (e.g. procedures) and working upwardly in the SDL hierarchy.

If you work bottom-up, select the *Down* option to enable access to signals used one level down in the SDL hierarchy. This option is enabled by default.

– You may also type `<Ctrl+d>` to toggle the option. Typing `d` brings the *Down* option into view (if enabled).

Working with Local Signals

If you look for entities that are used locally in a diagram, select the *This* option. Remember, a diagram needs to be saved in order to update the signal dictionary.

– You may also type `<Ctrl+t>` to toggle the option. Typing `t` brings the *This* option into view (if enabled).

– To find the definition of the signal in the *This* section, you may select an object in the drawing area that uses the signal and type `<Ctrl+f>`.

Listing all Signals

You can list all signals that are visible according to the SDL scope rules by turning the *All* option on. For instance, a signal that you declare on the system level will be available in all diagrams in the SDL hierarchy.

– You may also type `<Ctrl+a>` to toggle the option. Typing `a` brings the *All* option into view (if enabled).

Using Packages

If you want to gain access to signals that you have declared in a package, you should enable the *All* option.

Using Diagram Types

If you use the diagram type concept, you should enable the *All* option to gain access to the signals that are defined in the diagram types.

Listing MSC Messages

If you describe the dynamic behavior of an SDL diagram using the MSC Editor, you may take advantage of this by turning the *MSC* option on.

> **Note:**
>
> Messages used in an MSC will be available in the signal dictionary only if it is associated with an SDL diagram, i.e. linked into the SDL diagram structure.

Importing an External Signal Dictionary

You may import an external signal dictionary into the SDL Suite through the Postmaster interface. This is described in *"Load External Signal Definitions into the Information Server" on page 684 in chapter 12, Using the Public Interface*.

Locating the Source Diagram

Note: Signal Dictionary and OO

If you are looking for a source diagram that is an SDL-92 type (system type, block type, process type or service type) and that you access through an instance of the type, the following conditions must be respected in order to have the signal dictionary window list the diagram and list all of the signals and signal conveyors that are used in the diagram:

- The type diagram **must be referred in the same diagram as where it is instantiated**. In the signal dictionary window, only the type will be listed, not the instance. Figure 416 shows an example of this.
  - As a consequence of this, you will not be able to look into the instance of a type that is defined in a package diagram and used in the current SDL system.
- In the case the type is inherited from a supertype, the entities that you are looking for **must be defined in the child type**.

The main cause of this limitation is that channels and signal routes are connected to the instance, not to the type.

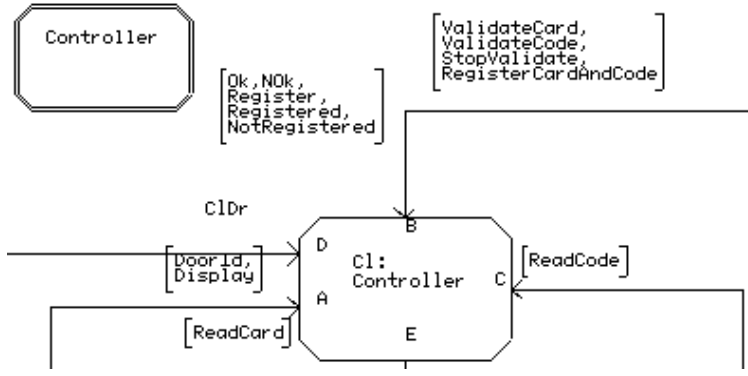


Figure 416: The type is referred where instantiated

The process type Controller is instantiated as Cl. The process instance Cl will be visualized as the process type Controller, and you can list signals conveyed on the gates A, B, C, D and E.

To locate the source diagram (i.e. the diagram where the signal is used or where the signal conveyor is available):

1. Locate the section (*Up*, *This*, *Down*, *All*, *MSC* and *External*) by scrolling up or down. The sections are listed in that order and the start of each section is identified with a separator with the corresponding name.

   – You may also use the keyboard accelerators `<Ctrl+u>`, `<Ctrl+t>`, `<Ctrl+d>`, `<Ctrl+a>`, `<Ctrl+m>` and `<Ctrl+e>` to toggle the respective option on and off.

   – Use the keyboard accelerators `u`, `t`, `d`, `a`, `m` and `e` to bring the respective section into view.

2. Identify the source diagram by its type and name. Diagrams are listed alphabetically by their name. The diagram type is identified either by a graphical SDL-like symbol (see Figure 424 on page 1997) in the left margin **(UNIX only)**, or by a text string that consists of the diagram type.

Updating the Text in the Target Diagram

Once you have located the source diagram your next task is to insert the text belonging to the object you have selected in the target diagram (the diagram where the change is to be done).

Updating a Signal / Timer Output, Input, Priority Input or Save

This operation makes sense only on pages where the implementation is described, i.e. the flow pages.

1. Locate the matching signal or timer in the signal list.

   – Signals are identified graphically with the SDL signal input and/or signal output symbol **(UNIX only)**, or with the characters ‘I’, ‘O’ or ‘?’ (see “Signal and Signal List” on page 1999.

   – Timers are identified with the MSC timer symbol **(UNIX only)**, or the character ‘T’ (see “Timer” on page 2001).

2. Make sure the signal output, input, priority input or save symbol to be updated is selected in the target diagram.

3. Select the signal or timer item in the signal list. Use *Insert* or *Replace* from the *Edit* menu to update the symbol contents. The target symbol is updated with the signal name and its formal parameters.

   – Alternatively, double-click the signal or timer.

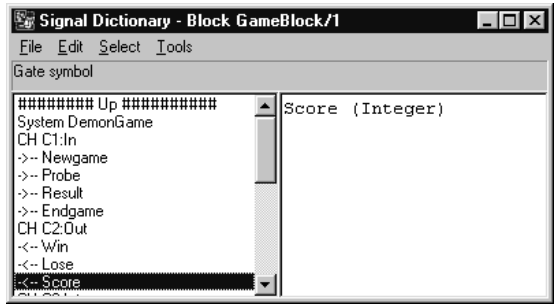


Figure 417: Selecting the signal Display

4. Update the signal's formal parameter type with a current parameter.

Updating a Signal Conveyor (Channel, Signal Route, Connection Point or Gate)

1. Locate the signal conveyor that constitutes the corresponding item that interfaces to the channel, signal route or gate that you are to update in the current diagram.

   – In graphical notation **on UNIX**, these entities are identified graphically by a symbol which has the appearance of the corresponding SDL concept (see Figure 425 on page 1998, Figure 426 on page 1998 and Figure 427 on page 1999).

   – Channels, signal routes and gates are identified textually with the abbreviations CH, SR and GA, respectively.

2. Make sure the **signal list** is selected in the target diagram.

3. Select the signal conveyor. Use the *Insert* or *Replace* from the *Edit* menu to update the signal list contents.

   – Alternatively, double-click the signal or timer.

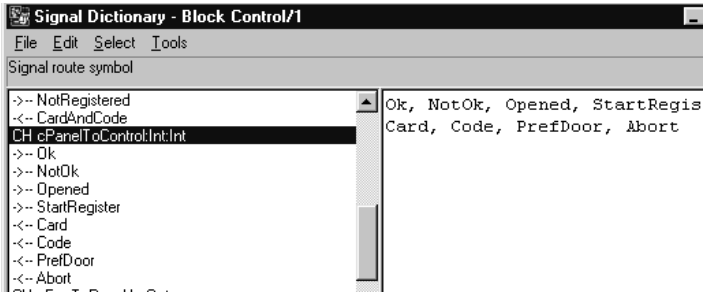


Figure 418: Selecting the Channel cPanelToControl:int:int

4. In the case of a bidirectional conveyor, you will see lines of text in the right window, each one corresponding to a signal list (as in Figure 418, above). You should copy each line of text to the corresponding signal list. Select the signal list in the SDL Editor and a line, then use *Insert* or *Replace* to update the signal list.

Updating a Text Symbol

When updating a text symbol, you are likely to declare signals and timers that you have referred to in the current diagram or referred to in the SDL hierarchy descending from the current diagram.

1. You should make sure the *Down* and *This* options are selected.
2. Remember to add a SIGNAL or TIMER keyword in the text symbol, before the enumeration of signals.
3. Locate all diagrams of interest. Look for signals / timers that you want to declare. Double-click each signal, and insert the necessary commas to create a syntactically correct declaration. Terminate the operation by inserting a semicolon.

Finding a Definition

When you select an object in the SDL Editor's drawing area and then type `<Ctrl+f>` the first occurrence of the first word in the object is looked for and selected in the This section.

Closing the Signal Dictionary Window

You close the signal dictionary window by selecting *Close* from the signal dictionary *File* menu.

The Symbol Label

The *symbol label* field is a non-editable text field that displays the type of object that is currently selected in the SDL Editor.

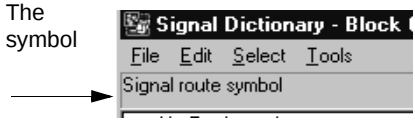


Figure 419: The symbol label

The symbol label reads “No single symbol selected” in the following circumstances:

- No object is currently selected in the SDL Editor
- More than one object is selected.
- The symbol type is not defined in the Grammar Help file.

The Signal List

The *signal list* is a selectable list where all occurrences of signals that are found according to the current selection criteria are listed.

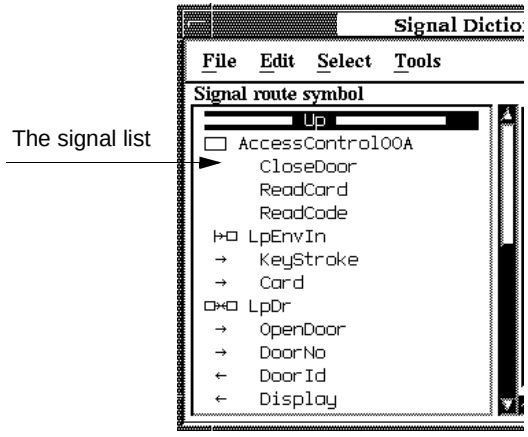


Figure 420: The signal list **(on UNIX)**

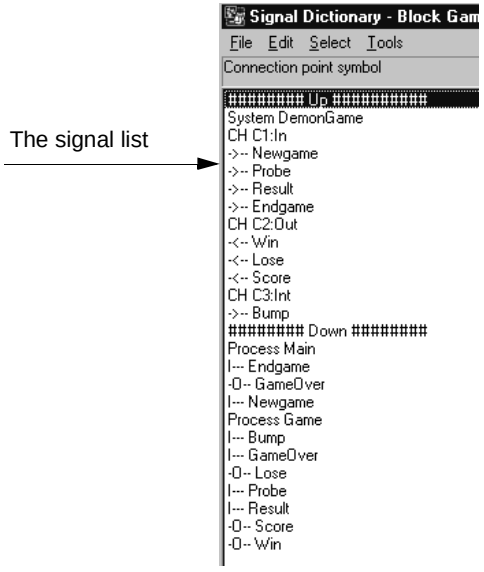


Figure 421: The signal list **(in Windows)**

The signal list is empty when:

- The symbol label reads “No single symbol selected”
- The signal dictionary function does not make any sense on the selected object. i.e. the selected object does not refer/define a signal.

The following objects are recognized as carrying signal information:

Object	Usage
channel line channel name signal list in channel	Reference to signal
connection point	Reference to signal
gate gate name signal list in gate	Reference to signal
input	Reference to signal
output	Reference to signal
priority input	Reference to signal
save	Reference to signal
signal route line signal route name signal list in signal route	Reference to signal
text	Declaration of signal

Signal List Notation and Syntax

Each line in the signal list follows a notation that informs about the various diagrams and symbols where signals are used and also about the context.

In Windows, this notation is purely textual.

On UNIX, the notation that is adopted is an SDL-like graphical notation. It is also possible to select a textual notation if the font face used in the graphical notation is not supported by the computer. You select the textual notation by setting the environment variable `SDLENOGRAPHICS` before you start the SDL Suite .

Each item in the signal list is one of the following:

- *Separator*
- *Diagram Identifier*
- *Gate*
- *Channel*
- *Signal Route*
- *Signal and Signal List*
- *Timer*

Separator

The signal list is divided into several sections. The top of each section is clearly marked with a separator, having the appearance of a thick line (graphical notation **on UNIX**) or a number of hashes ("###...###") preceding and following a text that identifies the section.

Figure 422: A signal list separator (graphical notation **on UNIX***)*

```
######## Up ##########
```

Figure 423: A signal list separator (textual notation)

Each of these sections corresponds to an option that is enabled in the *Select* menu (see "Select Menu" on page 2031). Their meaning and order of appearance in the signal list is as follows:

- *Grammar*

  SDL Grammar Help section. See "Using Grammar Help" on page 1972 for more information about this topic.

- *Up*

  A list of all SDL signals that are available[1] by looking one level up in the SDL hierarchy. The *Up* section is empty when working on the top diagram (e.g. a system diagram).

- *This*

  All SDL signals that are available by looking at the current diagram, i.e. *this* diagram.

1. The term "available" means defined and / or referenced.

- *Down*

  All SDL signals that are available by looking at all diagrams one level down in the SDL hierarchy. The Down section is empty if the current diagram has no diagrams below it in the hierarchy.

- *All*

  All defined SDL signals that are visible for the current diagram according to the *SDL scope rules*.

- *MSC*

  Signals that have been exported from a Message Sequence Chart into the SDL Editor signal dictionary facility. An MSC must be *Associated* with an SDL diagram in the Organizer's diagram structure in order to include its signals into the Signal Dictionary (see "Associate" on page 98 in chapter 2, *The Organizer*).

- *External*

  Signals that have been imported from an external signal dictionary. An external signal dictionary is imported through the public PostMaster interface of the SDL Suite. See *"Load Definition File" on page 627 in chapter 11, The Public Interface*.

Diagram Identifier

In the graphical notation **on UNIX**, diagrams are identified with an SDL Reference symbol in the left margin which identifies the diagram type, followed by the diagram name:

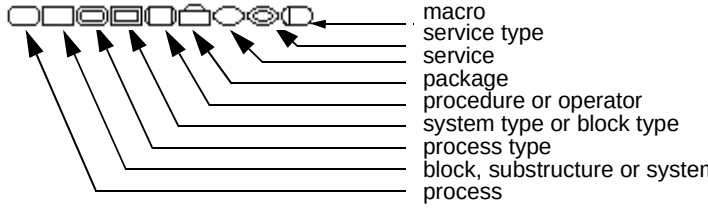


Figure 424: Diagram type symbols (graphical notation **on UNIX***)*

The textual notation reads `<diagramtype diagramname>`

All information that is listed after a diagram identifier is related to that diagram, until the next diagram identifier or separator.

Gate

In the graphical notation **on UNIX**, gates are identified by an arrow pointing into or out from the frame, followed by the name of the gate:

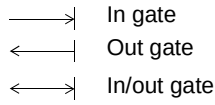


Figure 425: Gate symbols (graphical notation **on UNIX***)*

The textual notation reads:

```
GA <Gatename>:<In>|<Out>|<In:Out>
```

Signals on a gate are the lines that appear after the gate and before the next gate, channel, signal route or diagram identifier. The lines consist of all signals or signal lists that are conveyed on the gate, the signals sent to the diagram appearing first and the signals sent from the diagram appearing last.

Signals are listed according to the order of appearance in the respective signal lists.

If you select a gate in the list, one or two lines is displayed in the signal definition list. In the case of a bidirectional gate, the first line holds the signals sent to the diagram while the second line holds the signals sent from the diagram.

Channel

In the graphical notation **on UNIX**, channels are identified by an arrow between the frame and a block symbol (in the case of a channel to/from the environment), alternatively with an arrow between two block symbols (an internal channel):

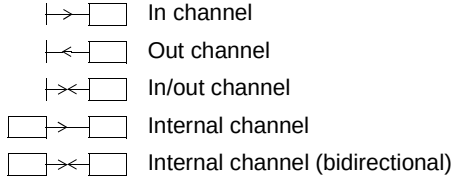


Figure 426: Channel symbols (graphical notation **on UNIX***)*

The textual notation of a channel is:

```
CH <Channelname>:<In>|<Out>|<In:Out>|<Int>|<Int:Int>
```

Channels are present on package, system, block, block substructure and system type diagrams only.

The lines of information that follow a channel consist of the signals that are conveyed on it. The appearance is similar to listing signals on a gate. See "Gate" on page 1998 for information about listing order.

Signal Route

In the graphical notation **on UNIX**, signal routes are identified by an arrow between the frame and a process symbol (a channel to/from the environment), alternatively with an arrow between two process symbols (an internal signal route).

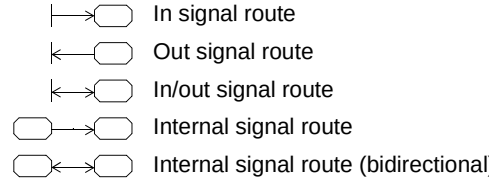


Figure 427: Signal route symbols (graphical notation **on UNIX***)*

The textual notation of a signal route looks like this:

```
SR <Signalroutename>:\
<In>|<Out]>|<In:Out>|<Int>|<Int:Int>
```

Signal routes are present on block and block type diagrams only.

The lines of information that follow a signal route consist of the signals that are conveyed on it. The appearance is similar to listing signals on a gate. See "Gate" on page 1998 for information about listing order.

Signal and Signal List

Signals and signal lists can appear in the following contexts:

- Signal Definitions (i.e. declarations in text symbols)
- Signal References on Lines (i.e. references in gates, channels and signal routes)

- Signal References in Flow Diagrams (i.e. references in flow diagrams; processes, process types and procedures).

The notation that is adopted differs slightly depending on the context.

A **signal definition** appears directly after the diagram identifier where the signal is declared. The signal is identified by its name and its location to distinguish it from other items.

The indentation is filled with blank space (see Figure 428).

```
System AccessControlOOA
     CloseDoor
     ReadCard
     ReadCode
```

Figure 428: Signal definitions

Note:

Each signal appears only once within a diagram. If a signal is referenced at least once within the diagram, it is not listed in the signal definitions section for that diagram, only references are listed.

A **signal reference** on a line uses a similar notation as for signal definitions. The indentation consist of an arrow that identifies the direction of the signals (see Figure 429).

```
lprc:Int:Int
-->- ValidateCard
-->- ValidateCode
-<-- StopValidate
```

Figure 429: Signal references on a line

Signal references in flow diagrams are listed along with the context the signal is used. This is indicated by the following attributes (multiple attributes can be set):

- Input[1]
- Output
- Reference to an undefined signal.

1. Save of signal is also considered as input by the Signal Dictionary function.

In the graphical notation **on UNIX**, these attributes are shown using SDL input and SDL output symbols, and a question mark for showing that the signal is not declared:

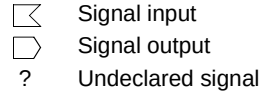


Figure 430: Signal symbols (graphical notation **on UNIX***).*

The textual notation uses the characters:

```
[I][O][?]
```

These characters appear at fixed positions. Where an attribute is not set, this is indicated with a hyphen (‘-’). See Figure 431.

```
IO?- Undefined
```

Figure 431: An undeclared signal which is input and output

Timer

A timer is handled identically as described in “Signal and Signal List” on page 1999. To indicate that the item is a timer, the graphical notation **on UNIX** uses the MSC symbol for timer.

Figure 432: Timer symbol (graphical notation **on UNIX***).*

In the textual notation, a ‘T’ appears as the rightmost attribute.

Timers can appear in the following contexts:

- Definitions of timers
- Set statements
- Reset statements
- Active statements.

Error Notification

If the SDL Suite fails in extracting signal information from some SDL diagrams, the information contained in that diagram is not available to the signal dictionary function.

In the graphical notation **on UNIX**, this is indicated in the signal list by a “bug” symbol located to the right of the diagram names.

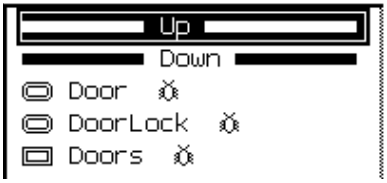


Figure 433: Errors were detected when extracting signals (graphical notation **on UNIX***)*

The SDL Suite failed in extracting signal information from the process type diagrams Door, DoorLock and the block type diagram Doors (one level down in the SDL hierarchy)

In the textual notation, this is indicated by an error message following the diagram name:

```
<diagramtype diagramname> :GrPrError
```

```
######## This ########
Process Type Door :GrPrErr
```

Figure 434: Errors were detected when extracting signals

The current diagram (This diagram) is process type Door. The SDL Suite could not convert this diagram to SDL/PR.

These error situations are either caused by:

- Syntactically incorrect SDL which causes the SDL Suite to fail when converting the SDL/GR diagram to SDL/PR.
- Syntactically incorrect SDL which causes the information server to fail when parsing the contents of the SDL/PR file. When possible, the signal definition field will contain a graphical reference to the source of error. Double-clicking that graphical reference will display the source of error in an SDL Editor window.

The error reports are listed in the Organizer low window. See also “Signal Dictionary Update” on page 1984.

The Signal Definition

The signal definition contains the following information:

- If a signal or timer is selected in the signal list, the signal or timer is listed with its parameters (see Figure 435).

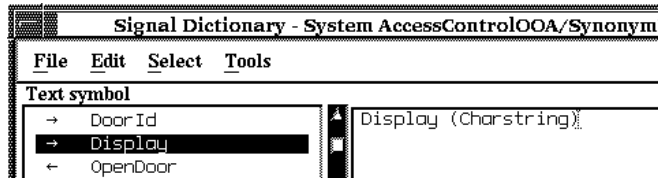


Figure 435: Selection of a signal

The signal Display is selected. The signal definition shows the signal along with its parameters - one parameter of type Charstring in the example.

- If a channel, signal route or gate is selected, the signal definition holds one or two lines of information. Each line contains a list of all signals that are conveyed (see Figure 436).
  - The first line contains the signal list to the parent diagram
  - The second line contains the signal list from the parent diagram.

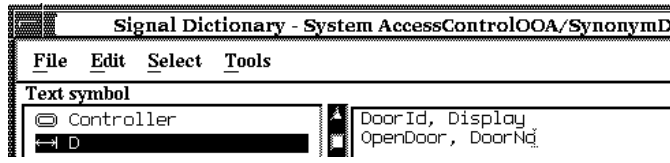


Figure 436: Selection of a gate

The gate D, which is an in/out gate is selected. The signal definition shows one line with all in-signals (Door, Display) and one line with all out-signals (OpenDoor, DoorNo) in relation to the parent diagram (Controller)

- If a signal list[1] is selected, the signal definition holds one line of information containing a list of all signals contained in the signal list. The syntax is

```
SignalListName = Signal1, Signal2,...,SignalN
```

1. A signal list is identified with its name within parentheses, such as `(SignalListName)`

Menu Bars

The SDL Editor's menu bar provides commands available in menus and menu choices for editing diagrams and pages of diagrams. Most of the functionality that the SDL Editor offers is contained within the commands from the menu bar. The menu bar provides the following menus:

- *File Menu*
- *Edit Menu*
- *View Menu*
- *Pages Menu*
- *Diagrams Menu*
- *Window Menu*
- *Tools Menu*
- *Help Menu*

The *File* menu is described in "File Menu" on page 8 in chapter 1, *User Interface and Basic Operations* and the *Help* menu in "Help Menu" on page 15 in chapter 1, *User Interface and Basic Operations*.

In addition, other menus may appear; one is the *Simulator* menu which appears when you set breakpoints in the SDL Editor (see *"Connect-To-Editor" on page 2150 in chapter 49, The SDL Simulator*).

Edit Menu

The *Edit* menu provides editing functions that you can perform on the objects in the drawing area. The *Edit* menu functions that are available differ depending on the type of diagram you edit, either:

- *Interaction Diagrams*
- *Flow Diagrams*
- *Overview Diagrams*

The *Edit* functions are as follows:

- *Undo*
- *Cut*
- *Copy*
- *Paste*
- *Insert Paste* (available on flow diagrams only)
- *Paste As*
- *Clear*
- *Collapse/Expand*

- *Redirect*
- *Bidirect/Unidirect*
- *Class...*
- *Flip*
- *Split Text*
- *Dash/Undash*
- *Mark as Important/Mark as Detail*
- *Mark Types...*
- *Drawing Size*
- *Select All*
- *Insert signal*
- *Complete Word*
- *Select Tail* (available on flow diagrams only)
- *Symbol Border Color > Set Color...*
- *Symbol Border Color > Set Default*
- *Symbol Border Color > <Color>*
- *Symbol Border Color > List All Colors...*
- *Symbol Fill Color > Set Color...*
- *Symbol Fill Color > Set Default*
- *Symbol Fill Color > <Color>*
- *Symbol Fill Color > List All Colors...*
- *Symbol Visibility > Hide*
- *Symbol Visibility > Show*
- *Include Expression*

Undo

With this command you restore the content of the drawing area to its state prior to the operation you performed most recently. It is only available for operations on symbols, lines and text attributes. You can undo the following operations:

- *Cut*, *Paste* and *Clear*
- *Flip*, *Redirect*, *Bidirect/Unidirect*
- *Adding Symbols*
- *Moving Symbols*
- *Resizing Symbols*
- *Drawing Lines*
- *Re-Routing and Reshaping Lines*
- *Resizing a Page*
- *Tidy Up*.

Note: Undo for text editing

You can undo text editing operations in the text window, such as *Import*, *Cut*, *Paste*, and *Clear*. There is also an *Undo* function available for text entered via the Grammar Help or Signal Dictionary functions. See "Undo" on page 2030.

Note: Undo for Class Symbols

If a class symbol is cut and thus also cleared, *Undo* will be enabled. If the Undo command is used, operators will be restored, including their body code.

Cut

This command removes the current selection from the drawing area or text window, and saves it in the clipboard buffer. Also see "Deleting an Object" on page 461.

In text editing mode the command will work on the current text selection.

Copy

This command makes a copy of the current selection, and saves it in the clipboard buffer.

In text editing mode the command will work on the current text selection.

Paste

This command inserts the contents of the clipboard buffer into the drawing area or text window.

No lines will be drawn between the pasted symbols and the symbols already present in the drawing area.

You can interrupt the *Paste* operation by pressing `<Esc>`.

Also see "Pasting an Object" on page 461.

In text editing mode the text contents from the clipboard buffer will be inserted at the current text insertion point.

Note: Cut, Copy and Paste for Class Symbols

When a class symbol is cut or copied, defined body codes are saved on the clipboard, along with their operators. When a class symbol is pasted, previously copied body codes are inserted as well, unless an operator already has defined another body code.

If any operator will be deleted as a result of the operation, a warning is shown, the command is canceled and a dialog similar to the one in "Cutting, Copying and Pasting Class Symbols" on page 1917 is shown.

Insert Paste

This function has the similar functionality as the *Paste* function, but is only available for **flow pages**. It inserts necessary space after the selected symbol before pasting the clipboard buffer and connecting the two flows.

The function requires that the clipboard is not empty and that you have selected exactly one symbol or line. Otherwise, the menu choice is dimmed. The menu choice is also dimmed if connecting the two symbol flows would violate the Z.100 syntax rules.

The functionality differs depending on if you have selected a symbol or a line. If you have selected a symbol the pasted symbols are inserted following the selected symbol. If for example you have selected a decision, the Insert Paste command will create a new branch. Also if a line following the selected symbol joins another line and the selected symbol only can have one follower, the inserted symbols will be placed following both the lines.

If you have selected only a line the pasted symbols will be inserted onto that line. For example if a line following a decision is selected, the inserted symbols will be inserted into the selected decision branch.

Before inserting the selection, the SDL Editor allocates the necessary space by pushing the required objects downwards. If this is not possible, a dialog is displayed.

- Clicking *Increase Page* in the dialog transfers control to the *Drawing Size* dialog, where you can specify the new width and height of the SDL page. The SDL Editor then carries on the operation and inserts the selection.

Paste As

Pastes the currently copied object (from the OM or Text Editor) as an SDL object in the drawing area. The object is transformed and a link is optionally created between the copied and pasted objects.

The *Paste As* dialog is opened. See "The Paste As Command" on page 448 in chapter 9, *Implinks and Endpoints* for more information.

Clear

This command removes the current selection from the drawing area or text window. The content of the clipboard buffer is not affected by this menu choice.

Also see "Deleting an Object" on page 461 in chapter 9, *Implinks and Endpoints*.

In text editing mode the current text selection will be removed.

> **Note: Deleting Class Symbols**
>
> When deleting a class symbol, a warning dialog is shown if an operator will be deleted as a result of this operation. See "Cut, Copy and Paste for Class Symbols" on page 2007.

Collapse/Expand

This command is available for symbols that can be clipped. These are the package reference symbol, additional heading symbol, text symbol, comment symbol and text extension symbol. A collapsed symbol has the minimum size and an expanded symbol will automatically adjust its size to the text.

Redirect

This command changes the direction of a selected channel, signal route or gate. The new direction is denoted by changing the orientation and position of the arrow, leaving the arrowhead pointing in the opposite direction. Subsequent redirections result in the signal list toggling between two directions.

Note:

The **only** way to draw a channel **from the environment to a symbol** is to:

1. Draw a channel from that symbol to the environment
2. *Redirect* the channel.

Bidirect/Unidirect

This command changes the selected channel, signal route or gate to be bidirectional/unidirectional. The Bidirect command will generate a new arrowhead and an empty signal list. The Unidirect command will show a dialog asking what signal list that should be kept.

Class...

This command opens up the *Browse & Edit Class Dialog*. The command is available only when a single class symbol has been selected. The command is not available if the class name contains incorrect syntax.

Flip

This command creates a vertically symmetrical copy of the symbol(s) you have selected. It is valid on the following symbols only:

- Input symbol
- Output symbol
- Priority input symbol
- Text extension symbol
- Comment symbol

The flow lines connected to a mirrored text extension symbol or comment symbol are redrawn in order to reconnect them to the opposite side.

Split Text

This command splits the text from a symbol and its associated text extension symbol (if any) into two parts, where the first part is placed in the symbol itself, and the second part in an associated text extension symbol. The split is done where the text insertion marker is, in the selected symbol.

To place all text in an associated text extension symbol, place the text insertion marker first in the text for a normal symbol and invoke this operation.

To remove an existing text extension symbol associated with a normal symbol, place the text insertion marker last in the text extension symbol and invoke this operation.

Dash/Undash

A dashed reference symbol is used in an SDL subtype when referring to an object that is defined elsewhere in one of the supertypes of the current subtype.

In a similar fashion, a dashed gate indicates that the gate is already defined in one of the supertypes in the inheritance chain.

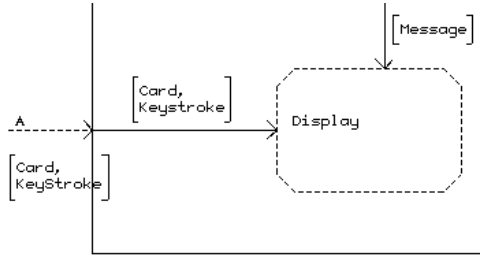


Figure 437: A dashed process reference symbol and a dashed gate

The *Dash/Undash* command is available on the following diagrams:

- *System Types*
- *Block Types*
- *Process Types*
- *Service Types*

The Undash is a toggle command.

Note that a symbol cannot be dashed if there are syntax errors in the name attached to the symbol. Also if a syntax error is introduced in a dashed symbol the symbol will automatically be undashed.

Mark as Important/Mark as Detail

In the high-level view, only symbols which are marked as “important” will be shown. For more information, see “Show High-Level View/Show Detailed View” on page 2015.

Symbols that are marked as “important” will be shown with border and fill colors different from ordinary symbols. These colors can be changed through the Preference Manager.

The Mark as Detail command sets the view status of a symbol to “normal”, meaning that it will only be shown in the detailed view.

Mark Types...

This command brings up a dialog with a list of symbol types. The selected symbol types will be visible in the high-level view, while unselected symbol types will only be visible in the detailed view.

Drawing Size

This menu choice sets the size of the *drawing area* for the current SDL page. A dialog is issued where you can type in the width and height. The values are saved on file when you save the diagram.

> **Note:**
>
> Enlarging the drawing, the current SDL page may not fit any longer into the paper that is defined with the Print preferences; the result may be an SDL page that requires multiple sheets of paper when printed.

Select All

This command selects all of the text contained in the text window or all objects within the drawing area.

Insert signal

When an input or output symbol is selected this menu-choice will show a sub-menu with a list of signals available to the system. If an input symbol is selected it will list input signals and if an output symbol is selected output signals will be listed. Selecting a signal will place it in the symbol.

Complete Word

> **Note:**
>
> You can also complete a word by having the mouse pointer in the drawing area and pressing the tab key, provided that the preference Editor**TabEqualsCompletion* is true.

Word completion helps you complete long words that have already been used in the same context. Type the beginning of the already used word and invoke word completion to get the rest of the word. If there are several ways to complete the word, the word completion operation will initially give you the common part of the possible completions. If you invoke word completion again, you will get one of the alternatives. To get another alternative, invoke word completion once more.

Word completion takes the word closest to the current cursor position and tries to complete that word by adding characters at the end of the word.

The word completion operation uses the following sources to look for already used words:

- The current diagram page.
- Pages in the beginning of the diagram. The number of pages that the word completion operation examines is determined by the preference Editor**CompletionMaxPages*.
- The names of entities listed in any running index viewer. This makes it possible to reuse names from other diagrams than the current diagram.
- All words listed in the file .../sdt/include/keywords/sdl.txt in the installation. As default, this file contains SDL keywords.

Select Tail

This command selects the currently selected symbol and all objects beneath it within the tree (following the flow lines).

Symbol Border Color > Set Color...

This command calls the color dialog where it is possible to set a custom color for the selected objects.

Symbol Border Color > Set Default

This command restores the default color for the selected objects. Default color for Symbol borders are set by the Preferences.

Symbol Border Color > <Color>

This command sets <Color> for the selected objects. Default available choices are Black, Blue, Green, Cyan, Red, Magenta, Yellow, and White. Custom colors are added to the menu when they are first used. The most recently used color is moved to the top of the list.

Symbol Border Color > List All Colors...

This menu choice becomes available when a maximum of 20 used colors has been surpassed. When you select *List All Colors* a dialog is issued containing all colors that has been used in the current SDL Editor session and you have the possibility to set them for the selected objects.

Symbol Fill Color > Set Color...

This command calls the color dialog where it is possible to set a custom color for the selected symbols.

Symbol Fill Color > Set Default

This command restores the default color for the selected symbols. Default fill color is set by the Preferences.

Symbol Fill Color > <Color>

This command sets <Color> for the selected symbols. Default available choices are Black, Blue, Green, Cyan, Red, Magenta, Yellow, and White. Custom colors are added to the menu when they are first used. The most recently used color is moved to the top of the list.

Symbol Fill Color > List All Colors...

This menu choice becomes available when a maximum of 20 used colors has been surpassed. When you select *List All Colors* a dialog is issued containing all colors that has been used in the current SDL Editor session and you have the possibility to set them for the selected symbols.

Symbol Visibility > Hide

This command hides the selected symbols. The hidden symbols will be replaced with a thick vertical line. The flow line structure will not be altered. The [Screen, Print]Z100Symbol preferences makes the replacement line as this as the flow line. See also *Show hidden symbols* and *Include Expression*.

Symbol Visibility > Show

This command restores the selected symbols.

Include Expression

This command displays a dialog, where it is possible to specify the include expression that is presented above the top left corner of a symbol, for the selected symbols.

The include expression is a name of a boolean variable (or external synonym). The value of the include expression is checked before analyzing an SDL system, when SDL/GR is converted to SDL/PR. Only those symbols that...

- have no include expression
- have an include expression that evaluates to true

...are included in SDL/PR and are visible for the analyzer and the code generator.

As default, all boolean variables have a value of false. To change a boolean value to true, an external synonym file (a plain text file with extension *.syn) must be created and placed close to the SDL system in the Organizer. The external synonym file closest to the SDL system will be used. An external synonym file contains lines of name value pairs. For instance:

```
DEBUG true
VERSION1 false
VERSION2 true
```

Note that all lines going to a symbol that is removed, will also be removed. An exception to this rule is that flow lines going to a symbol that will be removed, are reconnected to the first following symbol that will not be removed, if there is one and only one such symbol.

It is also possible to decide if SDL/GR symbols should be included or excluded in SDL/PR by hiding symbols. See "Analyze" on page 112 in

chapter 2, *The Organizer*, Symbol Visibility > Hide and Symbol Visibility > Show.

View Menu

The *View* menu provides rescaling functions and access to various options that affect the behavior of the SDL Editor.

The following menu choices are available in the *View* menu:

- *Set Scale*
- *Show High-Level View/Show Detailed View*
- *Temporary Colors > Remove*
- *Temporary Colors > Make Permanent*
- *Window Options*
- *Diagram Options*
- *Editor Options*

Set Scale

Issues a dialog where you can adjust the scale.

Show High-Level View/Show Detailed View

The high-level view shows only symbols which are marked as “important”, whereas the detailed view shows all symbols (normal view). This makes it possible to work with a less detailed version of the SDL diagrams.

By selecting the high-level view, only symbols which are either marked as “important”, or belong to one of the types selected in the Mark Types dialog, will be visible. Also, a tidy up operation will be performed, and the diagram will be read-only - all editing has to be done in the detailed view.

By selecting the detailed view, the diagram is reverted to the original, detailed view, where all symbols can be seen and edited.

Temporary Colors > Remove

Temporary colors are colors that are not saved in the SDL diagram. These colors are used by certain operations, such as SDL trace from SDL simulators and *Compare Diagrams* in the SDL editor, to highlight symbols of interest. Before doing a new operation that relies on color,

it might be useful to remove already existing temporary colors. This can either be done by using this menu choice or by closing down the SDL editor.

Temporary Colors > Make Permanent

Temporary colors are explained in *Temporary Colors > Remove*. This menu choice is used to convert temporary colors into permanent colors that can be saved in diagram files for future use.

Window Options

This menu choice issues a dialog where you set the options that affect the window properties.

- *OK* applies the options as defined in the dialog to the current window only.
- *All Windows* applies the options as defined in the dialog to all windows opened by the SDL Editor.
- *Tool Bar* determines whether the tool bar should be displayed or not.
- *Status Bar* determines whether the status bar should be displayed or not.
- *Symbol Menu* determines whether the symbol menu should be displayed or not.
- *Text Window* determines whether the text window should be visible or not.
- *Page Breaks* determines whether *physical page breaks*, with the appearance of dashed horizontal and vertical lines, should be displayed or not in the drawing area. These page breaks are defined by the print preferences; they show where the print utility will break the diagram into multiple printout pages.
- *Show Grid* determines whether the current symbol grid points are shown as small markers.

Diagram Options

This menu choice issues a dialog where you set the options that affect the editing properties applied on the current diagram.

Interaction and Flow Diagram Options

A dialog with the following options is displayed on interaction and flow diagrams.

- *Use Grid*

  This option determines whether or not the symbols that are added, resized or moved should adhere to the symbol grid or not. By default the symbol grid is active.

  By disabling the symbol grid, the SDL Editor will use the minimum grid of 5 * 5 mm.

- *Layout syntax check*

  This option determines whether the SDL Editor should adhere to the formal SDL syntax rules regarding symbols and the connection of symbols. Ideally the layout syntax check should always be on.

  – Turning syntax checking *off* allows you to insert any symbol displayed in the *symbol menu* into the drawing area. In this mode, symbols can be interconnected in any manner. Furthermore, the connector symbol will always be treated as an inconnector in this mode. Therefore the Analyzer will detect errors when trying to analyze the diagram. To avoid problems turn off the syntax checking only if you want to draw diagrams that never should be analyzed.

Note:

Turning the switch from off to on will **not** perform a retroactive syntax check for not allowed or interconnection of symbols. Only the objects that you add while syntax checking is enabled are checked.

- *Textual syntax check*

  This option determines whether the SDL Editor should perform a syntax check on all the texts in the diagram. Ideally the textual syntax check should always be on.

  – Turning syntax checking *off* disables the textual syntax check except for the kernel heading and the reference symbols which are always checked.

- *Additional Heading only on first page*

This option determines whether the SDL Editor should present the Additional Heading symbol on each page. When the option is on, only the first page in the diagram will show the Additional Heading symbol.

The diagrams options are also saved on file when the diagram is saved.

Overview Diagram Options

A dialog with the following options is displayed on *overview* diagrams.

- *Show line names*

  This option governs whether the names of lines (such as channels and signal routes) should be displayed or not.

- *Show signal lists*

  This option governs whether the signal lists associated to channels and signal routes should be displayed or not.

The diagrams options are also saved on file when the diagram is saved.

Editor Options

This menu choice issues a dialog where you can customize the behavior of the SDL Editor.

The options are controlled by toggle buttons. They are:

- *Always new Window*

  This option indicates whether or not a new window should be opened whenever you select the *New* or *Open* command or any command from the *Pages* menu.

  The default behavior is not to open a new window.

- *Sound*

  This option indicates whether or not improper actions in the SDL Editor, such as attempting to overlap symbols, should be brought to your attention by producing an alert sound.

  The default value for this option is on.

- *Show link endpoints*

This option indicates whether endpoint markers should be displayed for symbols having link endpoints. Regardless of the setting of this option, link endpoints are never shown when printed.

- *Use tab key for word completion*

  See "Complete Word" on page 2012.

- *Show hidden symbols*

  This option indicates whether hidden symbols should be displayed as a vertical line. If this option is selected hidden symbols will be displayed normally but filled with a grid pattern.

- *Use informal view*

  This option sets the editor in Informal mode. In informal mode the content of the symbols is substituted with the content of the attached comment symbol, if any. The comment symbols are hidden. Symbols without an attached comment symbol are displayed normally.

- *Show signal declaration*

  This option decides if the signal declaration including parameters should be shown in the message area when a signal name is selected by clicking on it in the drawing area. Turn this option off if you think it takes too long for the editor to find the signal declaration. There is a preference deciding the default value of this option: Editor*ShowSignalDeclaration.

Pages Menu

The *Pages* menu contains the following menu choices:

- *First*
- *<Page Name>*
- *Last*
- *Add*
- *Edit*
- *Edit Reference Page*

First

This menu choice opens the first page contained in the diagram. The first page is defined according to the order of appearance in the *Edit Pages* dialog. If the page is already opened, its window is displayed.

<Page Name>

Activating the *Page* menu presents up to four menu choices that consist of the names of the two pages that are sequentially right before and after the page you edit. If you edit the first page of the diagram, the next four sequential pages are shown. If you edit the last page of the diagram, the previous four pages are shown.

When you select one of these page names, that page is opened or restored in the SDL Editor. Use the *Edit* menu choice if you want open other pages.

Last

This menu choice opens the last page of a diagram. See "First" on page 2019 for more information.

Add

This menu choice is a shortcut for adding one page to the current diagram. The *Add Page* dialog is issued and when you click *OK* the new page is shown.

Edit

This menu choice opens a dialog where you can *Add*, *Rename*, *Move up*, *Move down*, *Clear*, *Cut*, *Copy* and *Paste* an SDL Page. See "Page Editing Functions" on page 2033.

Edit Reference Page

This menu choice opens (or, if already active, simply restores) the page in the parent diagram where the name of the current page is referred.

This menu choice is dimmed when the current page is one of the pages contained in the Organizer *Root Document* (in which case there is no parent diagram) or the diagram being edited is not known by the Organizer.

Diagrams Menu

The *Diagrams* menu records all diagrams and pages that are opened by the SDL Editor.

- *Back*

- *Forward*
- *<Diagram Name>*
- *List All*

Back

This menu choice opens the diagram that was previously visible in the same window.

Forward

This menu choice opens the diagram that was displayed in the window before you pressed the back button.

<Diagram Name>

The last edited page always goes to the top of the list, and subsequently moves the other diagrams and pages down a position. A maximum of 9 open pages can be shown. A tenth one will be put at the top of the list, but any subsequent opening of a diagram or page will only show the last 9 that have been opened. Another option – *List All* (at the bottom of the list) – is available to list all the open diagrams in the SDL Editor.

Each item in the menu provides information about the diagram type, its name, a slash (‘/’) followed by a page name, a hyphen and, possibly, the file it is stored on. A diagram that is preceded by an asterisk (‘*’) denotes that it has been modified during the SDL Editor session.

List All

This menu choice becomes available when a maximum of 9 open pages has been surpassed. When you select *List All* a dialog is issued containing all diagrams and pages that are currently open in the SDL Editor and you have the possibility to display them.

Window Menu

The *Window* menu contains the following menu choices:

- *New Window*
- *Close Window*
- *Grammar Help*
- *Signal Dictionary*
- *Class Information*

- *Entity Dictionary*
- *<Window Name>*
- *List All*

New Window

This command opens a new window containing a new view on the SDL page contained in the source window from where you selected this menu choice. You can edit the SDL page in any window.

Close Window

This option closes the open window, but, does **not** close the diagram. All but the last open window can be closed, the last one you must close from the *File* menu, possibly in conjunction with saving information (see "Close Diagram" on page 14 in chapter 1, *User Interface and Basic Operations*).

Grammar Help

This menu choice opens the *Grammar Help* window, see "Grammar Help and Signal Dictionary" on page 1971. Its purpose is to assist you in editing SDL textual elements that are correct according to the SDL grammar.

Each SDL Editor window has its associated Grammar Help window.

The SDL Editor tries to open a file with grammar help templates when opening the Grammar Help window (which file to open by default may be specified as an SDL Editor preference). If the file cannot be located, a dialog is issued where you can choose to select it manually. The filter in the file selection dialog will be set to `*.tpl`.

Signal Dictionary

This menu choice opens the *Signal Dictionary* window, see "Using the Signal Dictionary" on page 1983. It gives access to information about what signals are defined in an SDL hierarchy. Each SDL Editor window has its associated Signal Dictionary window.

Class Information

Opens the *Class Information* window. See "Class Information" on page 1942 for more information.

Entity Dictionary

Opens the Entity Dictionary window. See "The Entity Dictionary" on page 434 for more information.

<Window Name>

If more than one editor window is open the other windows are listed here. The behavior of this list will be similar with the diagrams list in the *Diagrams* menu. The only difference is that the menu items will not provide the diagram file information.

List All

This menu choice is available only if more than 9 editor windows are open, and has the same functionality as the *List All* menu command in the *Diagrams* menu.

Tools Menu

The *Tools* menu contains the following menu choices:

- *Show Organizer*
- *Link > Create*
- *Link > Create Endpoint*
- *Link > Traverse*
- *Link > Link Manager*
- *Link > Clear*
- *Link > Clear Endpoint*
- *Search*
- *Spelling > Comments*
- *Spelling > All Text*
- *Tidy Up*
- *Connect to Text Editor*
- *Navigate*
- *Show > Definition of X*
- *Show > Use of X*
- *Show > Definition or Use of X*
- *Show > Next (Definition or Use of) X*
- *Show > Definition of X in Index Viewer*
- *Show GR Reference*
- *Create Bookmark*
- *Compare Diagrams*
- *Merge Diagrams*

- *Add Differences*
- *Generate PR*

Show Organizer is described in "Show Organizer" on page 15 in chapter 1, *User Interface and Basic Operations* and the Link commands are described in "Link Commands in the Tools Menus" on page 442 in chapter 9, *Implinks and Endpoints*.

Search

This menu choice opens a dialog that allows you to search for a text in the current diagram or document in any editor. You can search many diagrams and documents at the same time with this menu choice.

For more information about searching, see "Search" on page 142 in chapter 2, *The Organizer* and "Searching and Replacing Text" on page 1962.

Spelling > Comments

Check the spelling of comments in the current diagram. For more information about the spelling operation, see "Spelling > Comments" on page 147 in chapter 2, *The Organizer*.

Spelling > All Text

Check the spelling of all text in the current diagram. For more information about the spelling operation, see "Spelling > Comments" on page 147 in chapter 2, *The Organizer*.

Tidy Up

This menu choice rearranges the graphical layout of the entire SDL diagram, using default layouting algorithms when calculating the location and size of graphical objects. Before the tidy up operation is started, a dialog appears where you can adjust parameters to the operation:

- *Start transition on new page.* Decides if a state symbol representing the beginning of a new transition should be placed on the same page as already placed symbols, or on a new page.
- *Use extra space after comment and text extension.* Decides if a symbol in a flow, after a symbol with an associated comment or text extension, should be moved below the associated symbol or not.

- *Format input and output text.* Decides if text in input and output symbols should be reformatted or not. If the text is reformatted, the signal name is placed in the symbol, while any parameters are placed in a text extension symbol.
- *Replace textual comments with comment symbols.*
- *Place comment symbols to the left.*
- *No text extension for state.* Decides if text extensions should be used or not if the state name does not fit in the state symbol.
- *No text extension for decision.* Decides if text extensions should be used or not if the decision text does not fit in the decision symbol.
- *Sort states and transitions.* Decides the order state symbols (More precise: state and input symbol combinations) will be presented in the resulting diagram.

Connect to Text Editor

This command issues an external text editor and creates a temporary file from the currently selected text. After this you can only edit the text from the external editor. The SDL Editor is updated every time the external text editor saves the temporary file. When you no longer edit the temporary file the editing control returns to the SDL Editor.

Which external text editor you are to use is defined by the preference SDT*TextEditor.

Navigate

This command navigates to another diagram or to another symbol within the current diagram, depending on the selected symbol. It is available for the following symbols:

- For a reference symbol it will show the diagram being referenced.
- For an instantiation symbol it will show the corresponding type diagram. If there is more than one diagram that matches the type name, a dialog will appear where you can select one of the diagrams.
- For a create request symbol the process diagram for the created process will be shown. If the diagram is ambiguous a selection dialog appears.

- For a procedure call symbol the procedure diagram will be shown. If the diagram is ambiguous a selection dialog appears.
- For a macro call symbol the macro definition will be shown.
- For a state symbol another state or nextstate symbol using the same state name in the same diagram will be shown. A selection dialog will appear if there are more than one occurrence of the name.
- For an inconnector symbol the corresponding outconnector symbol will be shown. If more than one outconnector exists a selection dialog appears.
- For an outconnector symbol the corresponding inconnector symbol will be shown.
- For a class symbol, another class symbol, using the same class name in the same diagram, will be shown. If there is more than one occurrence of the name, a select dialog is shown.

Double-clicking a symbol will have the same effect as using the *Navigate* command. This applies to all navigable symbols, except class symbols.

The *Show* Sub-menu

All menu choices in the *Show* sub-menu require a running index viewer loaded with an up-to-date cross reference file. You can get this for all SDL systems that passes through the SDL Analyzer, by pressing the *Generate Index* quick-button in the Organizer.

Show > Definition of X

This command takes the name of the word closest to the current cursor position as input. If the name represents an SDL entity listed in the index viewer, the SDL Editor will navigate to one of the definitions of that entity. Usually there is only one definition to show, but for states, each state symbol is considered to be a part of the definition and one of them is shown. If you invoke this command once more for the same state name, another state symbol is shown (if there are several).

Show > Use of X

This command takes the name of the word closest to the current cursor position as input. If the name represents an SDL entity listed in the in-

dex viewer, the SDL Editor will navigate to one of the uses of that entity. If you invoke this command once more for the same word, another use of that entity is shown (if there are several uses),

Show > Definition or Use of X

This command takes the name of the word closest to the current cursor position as input. If the name represents an SDL entity listed in the index viewer, a dialog is opened, allowing you to navigate to one of the definitions or uses of that SDL entity.

Show > Next (Definition or Use of) X

This command repeats the last *Show > Definition of X*, *Show > Use of X* or *Show > Definition or Use of X* command using the same word as last time as input.

Show > Definition of X in Index Viewer

This command takes the name of the word closest to the current cursor position as input. If the name represents an SDL entity listed in the index viewer, the index viewer tool is popped up and this entity is selected.

Show GR Reference

This command issues a message where the graphical SDT reference for the object you have currently selected is displayed.

The syntax of the graphical references used in the SDL Suite environments described in chapter 18, *SDT References*.

Create Bookmark

This command creates a bookmark in the Organizer for the object you have currently selected.

Compare Diagrams

With this menu choice you can compare the contents of two SDL diagrams. See “Comparing and Merging Diagrams” on page 2036.

Merge Diagrams

With this menu choice you can compare the contents of two SDL diagrams and create a new diagram by merging the two SDL diagrams. See "Comparing and Merging Diagrams" on page 2036.

Add Differences

With this menu choice you can merge two SDL diagram versions into one. The SDL diagram that is visible in the editor when the operation is invoked will be updated with information from the other version, for all differences. Symbols from the other diagram might be placed slightly to the left of the original place, to avoid overlapping symbols.

Differences are marked with temporary color, except for complete page differences when a complete page is missing in the other diagram. The reason for this is to support merging of a complete diagram (the SDL diagram in the editor) with diagram parts (containing a subset of all pages in the complete diagram) that might be slightly different.

In the *Add Differences* dialog, it is possible to specify the other diagram to compare with, either by specifying a file name or by specifying another editor buffer. The *Add Differences* dialog also makes it possible to specify the temporary colors to use for highlighting differences.

Generate PR

This command will generate SDL/PR/CIF for the current diagram. A dialog with the following options appears:

- *PR file*

  The name of the PR/CIF file to create.

- *Generate GR references*

  This option will generate GR references that can be used to trace the PR code back to the GR diagram.

- *Generate CIF*

  CIF is the standard defined in ITU recommendation Z.106 to be used when transferring SDL/GR diagrams preserving the graphical layout. With this option set, CIF comments will be generated together with the PR.

The *File* Menu of the Text Window

The *File* menu provides functions that transfer text from a file to the text window and vice-versa. The basic intention is to provide you the possibility to edit larger portions of text with a more suitable text editor (for instance signal definitions in text symbols). Another possibility to edit text externally is to use the *Connect to Text Editor* command in the *Tools* menu.

Import

Import imports the contents of a file into the text window and inserts the contents of the file at the text cursor position, possibly replacing selected text in the text window. In the file selection dialog, the filter is set to `*.txt` by default.

Export

Export exports the selected text to a file. If no text is selected in the text window, the entire text window contents will be exported to the file. In the file selection dialog, the filter is set to `*.txt` by default.

Menu Bar in Grammar Help Window and Signal Dictionary Window

These menu bars contains the following menus:

- *File Menu*
- *Edit Menu*
- *Select Menu*
- *Tools Menu*

File Menu

The *File* menu contains menu choices that access files that contain *SDL Grammar Help templates*.

These commands are useful only when requesting Grammar Help support.

Load

This command opens a file selection dialog, where you can select and load the file with grammar help definitions.

Grammar template files normally have the file extension `.tpl`. With the SDL Suite, a standard template file, `sdt.tpl`, is enclosed. The SDL Editor attempts to locate a `.tpl` file each time you invoke the *Load* command.

Any template definitions that are already loaded will be replaced by the contents of the selected grammar help definition file.

Merge

Merge extends the current loaded grammar definitions with the contents of a grammar help definition file.

This command issues a file selection dialog, in which you can select and add a set of custom templates to the basic set provided in the SDL Suite.

Any template definitions that are already loaded will be extended with the contents of the merged template definition file.

Edit Menu

The *Edit* menu provides two utility functions that you can use when copying text from the *Grammar* window or the *Signal Dictionary* window to the SDL Editor's text window.

Undo

You can revert changes with the *Undo* command.

Insert

This menu choice copies the entire contents of the *Grammar* field or the signal definition and copies it into the text window at the current I-beam cursor location. Selected text will thus be overwritten.

A shortcut for *Insert* is to double-click on the template name in the name field.

Replace

This command replaces the contents of the text window with the contents of the *grammar field*. This function is a shortcut for the following sequence:

- "*Select All, Clear* and *Insert*" **(Grammar Help)**

- "*Select All* and *Insert*" **(Signal Dictionary)**

Select Menu

The commands on the *Select* menu define what functionality the *Grammar Help* window and the *Signal Dictionary* window should provide.

Each of the options can be individually turned on and off. In addition, an *Options* menu choice (see "Options" on page 2032) allows you to enable and disable multiple options in one command only.

The options that are currently enabled are indicated by an asterisk ('*') preceding the corresponding menu choice.

When you invoke the Grammar Help, the *Grammar* option is enabled, while the remaining options are disabled.

These functions restrict or extend the scope of search for signals. The Signal Dictionary functionality can be extended to provide access to MSC Messages and an external signal library.

Grammar

This toggle option, when set to on, will enable the SDL Grammar Help function.

Up

This toggle option enables the *Up* option provided in the Signal Dictionary window. This option displays the SDL signals that are available when looking one level up in the SDL hierarchy.

This

Turning *This* on lists all signals that are used in the current diagram.

Down

This option displays the SDL signals that are available when looking one level down in the SDL hierarchy.

All

This option displays all signals that are defined and are visible according to the SDL scope rules, starting from the root document (see "Root

Document" on page 42 in chapter 2, *The Organizer*) in the SDL System Structure chapter.

MSC

This option displays all messages that are available in the Message Sequence Charts that are submitted as input to the Signal Dictionary function. Each message is mapped to an SDL signal, with its parameters mapped to SDL signal parameters.

An MSC is submitted as input to the Signal Dictionary if associated to an SDL diagram that is present in the Organizer's diagram structure.

External

This option displays all signals that have been imported from an external signal dictionary. Importing external signals is done through the Public Interface. See *"Load Definition File" on page 627 in chapter 11, The Public Interface*.

Options

Issues a dialog where you can turn on and off all options individually (see "Select Menu" on page 2031 for a description of the options).

The default options for the Signal Dictionary are *Up* and *Down*.

Tools Menu

The Tools menu provide various convenience functions.

Show Editor

This command displays the parent SDL Editor window.

Show Definition

An SDL Editor comes up showing the definition of the item which is selected in the signal list.

Page Editing Functions

When you select *Edit* from the *Pages* menu, the *Edit Pages* dialog will be opened. In this dialog, you can add, rename, clear, cut, copy and paste an SDL page. The items in the dialog will be described below:

The Page List

This list includes all SDL pages in the SDL diagram. The operations available in the dialog, are only possible to perform if a page is selected in this list.

Edit

This button opens the selected page in the SDL Editor.

Cut

This button removes the selected SDL page from the diagram and saves it in the clipboard buffer. The information in the Organizer that is dependent on this page, will also be removed.

If the page contains reference symbols and these have an underlying structure in the Organizer this structure will be restored when you paste the page in a new position. See "Cutting, Copying and Pasting Reference Symbols" on page 1917.

The button is dimmed if the object you have selected is not an SDL page.

Copy

This button copies the selected page to the clipboard buffer.

The button is dimmed if the object you have selected is not an SDL page.

Paste

This button pastes a previously cut or copied page into the current diagram.

The *Paste* button will be dimmed if:

- The clipboard buffer is empty.
- The current clipboard contents would lead to a syntax violation in the target diagram.

The SDL Editor checks that the contents of the page are syntactically correct. If required a warning will be issued. Then a new dialog is displayed, where you should assign the page a new name. You can specify:

- The new name of the page (two pages with the same name are not allowed within a diagram)
- Possibly, an autonumbered name (see "The Autonumbered Option" on page 2036)
- If the new page should be pasted before or after the current page

Clear

This button removes the selected SDL page from the diagram. A confirmation dialog will be issued before the page is removed, and the SDL Editor will automatically rename autonumbered pages.

Caution!

As stated in the dialog, *Clear* on a page cannot be undone.

If the page contains reference symbols and these symbols have a substructure, this substructure will be removed from the Organizer. As there is no undo for the *Clear Page* command, a warning will be issued and it is possible to keep the substructure as a new hierarchy instead.

Move up

If an SDL diagram page is selected and you click the *Move up* button, the page will be moved up.

Note:

The undo function in the Organizer and in the SDL Suite will have no effect on page moves. It is, however, easy to undo a page move operation by just moving the page back again.

Move down

If an SDL diagram page is selected and you click the *Move down* button, the page will be moved down.

> **Note:**
>
> The undo function in the Organizer and in the SDL Suite will have no effect on page moves. It is, however, easy to undo a page move operation by just moving the page back again.

Add

This button opens a dialog where you can change some settings before new page is created:

- If autonumbering is turned off, you may specify any pagename, which must be in keeping with SDL conventions. If two pages are given the same name X the names will automatically be renamed to X_1 and X_2. The enumeration character can be chosen by the Editor preference Page*EnumerationCharacter.
- If autonumbering is on (the default), the page is automatically named using the next available sequential number within the diagram (1, 2, etc).
- If the page is to be inserted before of after the current page.
- If you add a page to a process or process type diagram, you can select if the new page should be a graph page (the default) or a service interaction page.
- If you add a page to a block or block type diagram, you can select if the new page should be a block interaction page or a process interaction page.

New pages are listed in the same order that they were added.

Rename

This button opens a dialog where you can rename the selected SDL page. If the page is part of a block or block type diagram, you can also change the page type (process interaction page or block interaction page) in this dialog.

Autonumbered pages cannot be renamed, unless you turn the option *The Autonumbered Option* off first.

The *Autonumbered* Option

If the option *Autonumbered* is on, a numeric name (1, 2, etc) will be applied to the selected page. In a dialog, you can select to auto-number the selected page or all pages in the diagram.

The *Open This Page First* Option

If the option *Open This Page First* is on, the selected page will be the page that will be displayed by default when you open the diagram. Only one page at the time can have this option set.

Comparing and Merging Diagrams

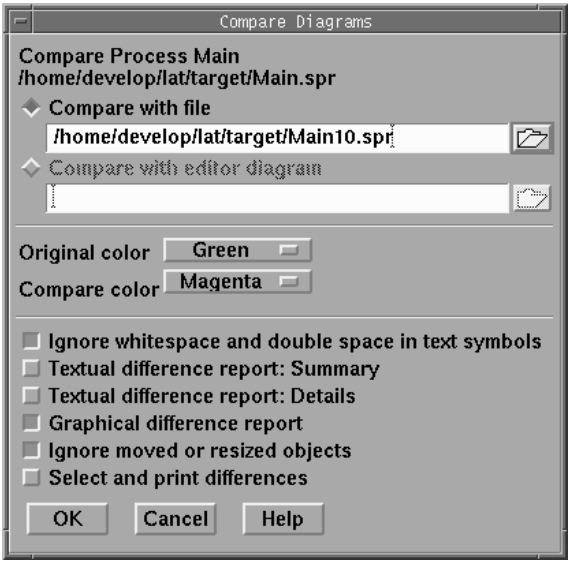


Figure 438: The Compare Diagrams dialog

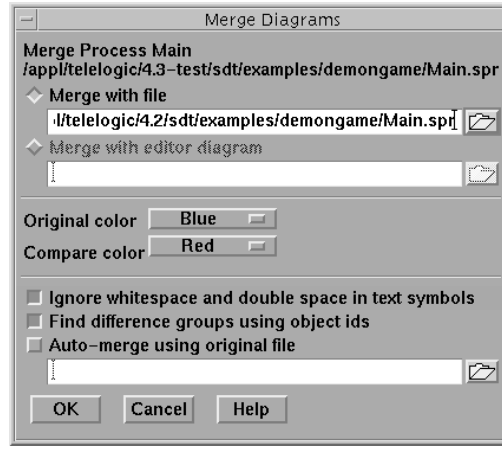


Figure 439 The Merge Diagrams dialog

When you select *Compare Diagrams* or *Merge Diagrams* from the *Tools* menu, a dialog will be opened where you can select the SDL diagram to compare with. (It is also possible to compare more than one diagram pair in one operation, but to do this you should use the corresponding menu choices in the Organizer. See "Compare > SDL Diagrams" on page 148 in chapter 2, *The Organizer* and "Merge > SDL Diagrams" on page 151 in chapter 2, *The Organizer*.

Specifying the Diagram to Compare With

There are two ways for specifying the diagram that should be compared with the currently displayed diagram:

- You may specify the file in the *Compare with file* field.
- You may specify another diagram already opened the SDL Editor in the *Compare with editor diagram* field. When you click the folder button, a dialog will be opened with a list of all diagrams opened in the SDL Editor (except for the currently displayed).

The option *Compare with editor diagram* has the two restrictions:

– It can only be used if more than one diagram is opened in the SDL Editor.

– The folder button can only be used if more than two diagrams are loaded into the SDL Editor.

Selecting the Report Type

In the Compare Diagrams dialog, there are six alternatives for reporting differences between diagrams: *Ignore white space and double space in text symbols*, *Textual Difference Report: Summary*, *Textual Difference Report: Details*, *Graphical Difference Report (with Merge Facility)* and *Select and Print Differences*.

The Merge Diagrams dialog does not allow you to get a textual difference report. Instead, you always get a graphical difference report and the only option available is *Auto-merge using original file*.

Ignore white space and double space in text symbols

When this option is on, differences caused by different usage of white space are ignored when comparing texts. One space character is considered equal to two space characters.

Textual Difference Report: Summary

This option is only available when comparing using the *Compare Diagrams* menu choice.

A summary of how many differences that has been found is presented in the Organizer log. The summary may look like this:

```
Comparing /home/lat/x.spr
     with /home/lat/y.spr
27 differences (16 symbols, 11 lines) :
  7 changed objects (7 symbols, 0 lines)
  13 removed objects (6 symbols, 7 lines)
  7 added objects (3 symbols, 4 lines)
```

When you read this report, notice the following:

- 27 is the total number of found differences in this diagram pair. This is the total of all changed, removed and added objects.

- An object is considered to be changed if you have moved or resized it, or if you have changed a text associated with the object.

If at least one of the two compared diagrams has more than one page, then a summary is also presented for each page pair. It may look like this:

```
Comparing /home/lat/version1/registeredcard.spt
     with /home/lat/version2/registeredcard.spt
5 differences (1 symbol, 4 lines) :
  1 changed object (1 symbol, 0 lines)
  2 removed objects (0 symbols, 2 lines)
  2 added objects (0 symbols, 2 lines)

Page details:

Page ValidateCard
5 differences (1 symbol, 4 lines) :
  1 changed object (1 symbol, 0 lines)
  2 removed objects (0 symbols, 2 lines)
  2 added objects (0 symbols, 2 lines)

Page RegisterMode
0 differences
```

Textual Difference Report: Details

This option is only available when comparing using the *Compare Diagrams* menu choice.

This option will print information about each found difference in the Organizer log. This makes it possible to navigate from the Organizer log to every found difference by using the *Show Error* operation in the Organizer log. See "Organizer Log Window" on page 183 in chapter 2, *The Organizer*.

The detailed textual difference report may look like this:

```
Comparing /home/lat/d1/db.sbk
     with /home/lat/d2/db.sbk

Comparing page 1

#SDTREF(SDL,/home/lat/d1/db.sbk(1),119(50,35))
ERROR This symbol has been changed...
#SDTREF(SDL,/home/lat/d2/db.sbk(1),119(50,40))
ERROR ...and now looks like this.

#SDTREF(SDL,/home/lat/d1/db.sbk(1),176(95,60))
ERROR This symbol has been removed.
```

```
#SDTREF(SDL,/home/lat/d2/db.sbk(1),182(60,60))
ERROR This line has been added.
```

Graphical Difference Report (with Merge Facility)

The merge facility is only available when merging with the *Merge Diagrams* menu choice, but you can get a graphical difference report both when comparing with the *Compare Diagrams* menu choice and when merging with the *Merge Diagrams* menu choice.

Two SDL Editor windows, named *Source 1* and *Source 2*, displaying the two diagram versions and their differences, and one dialog will be opened. For merge, there will be a third SDL Editor window, named *Result*, showing the merge result.

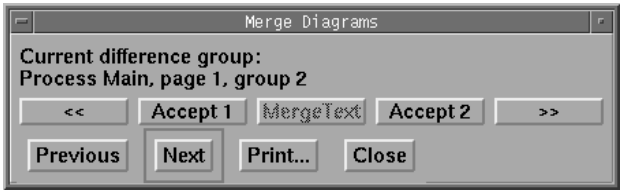


Figure 440: The Merge Diagrams dialog

The Compare Diagrams dialog is similar to the Merge Diagrams dialog. The difference is that the Compare Diagrams dialog is missing the first row of buttons. (“<<“, Accept1, Accept2 and “>>”.)

In the dialog that will be opened, you can:

– Navigate between difference groups with the *Next* and *Previous* buttons.

– (*Merge Diagrams* only.) Navigate between unresolved (not-yet merged) difference groups with the “<<“ and “>>” buttons.

– (*Merge Diagrams* only.) Do a merge, i.e. update the merge result diagram with information from the current difference group

and one of the compared diagram versions, with the *Accept1/Reject1* and *Accept2/Reject2* buttons.

- Compare or merge a text in a text symbol in the current difference group.
- Print the current difference found in the current page pair.

When the second compare dialog is closed, the editor windows are restored to how they appeared before the compare operation.

Comparing text in text symbols

When two text symbols differ the *CompareText* button will be available in the Compare Diagrams dialog and it will be possible to enter the Compare Text Symbols dialog.

The Compare Text Symbols dialog consists of two lists of text that are placed side by side with a "gutter" that will contain extra information about the differences. This gutter-information consist of:

- A "-" indicates that there is no corresponding line in the text compared to the other text.
- A "|" character indicates that this line differs in the two texts.

The current difference is indicated with a highlight. Buttons make it possible to navigate to *Previous* and *Next* difference item.

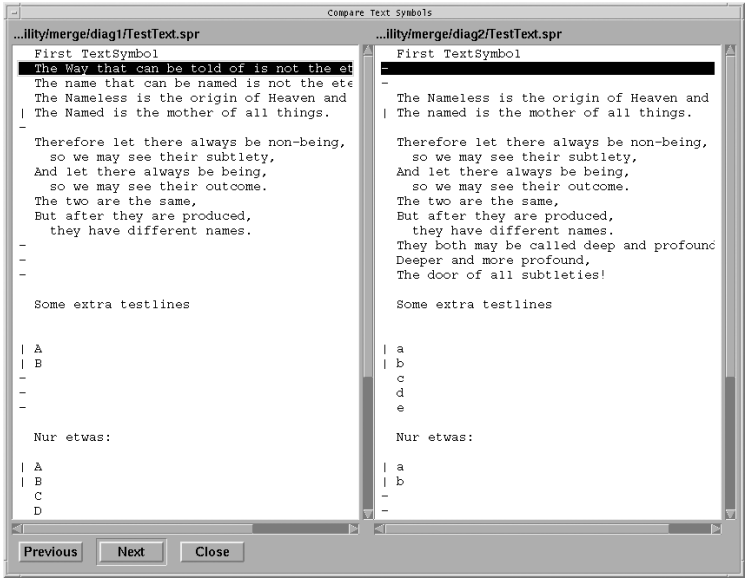


Figure 441: The Compare Text Symbols dialog

Merging text in text symbols

When manual merges are conducted the option to merge text symbols will always be available in the Merge Diagrams dialog. When merging a *MergeText* button is available as in the compare function. When a text symbol is selected there will be an option to select either symbol or merge the text in the symbols into a new symbol.

If the text merge is selected a Merge Text Symbols dialog is presented with three text lists, the two source texts and the result text. It is possible to navigate between the unresolved differences or between all differences.

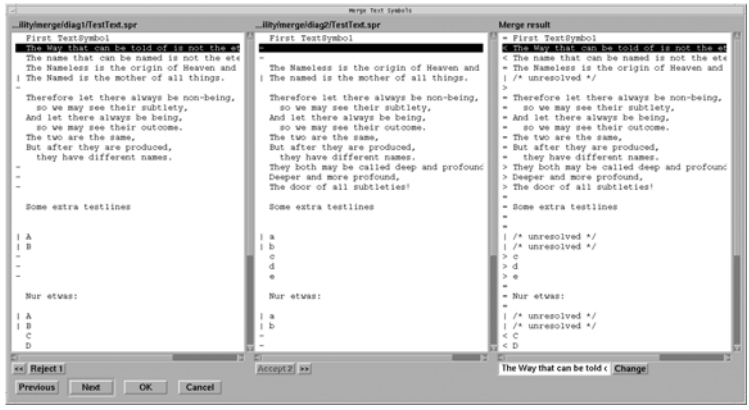


Figure 442: The Merge Text Symbols dialog

This works in the same way as in the normal merge. The buttons “<<“ and “>>” will jump between the unresolved differences and the *Previous* and *Next* buttons will jump between all differences.

The use of *Accept 1*/*Reject 1*, *Accept 2*/*Reject 2* buttons will select neither, either or both the texts for merge. The selections from the source files are presented in the result-list the gutter will show what version is selected. Additional symbols in the result list are:

- A “|” character indicates that this line differs in the two texts.
- A “=” character indicates that the selected text are equal in both versions.
- A “>” character indicates that the right text has been selected for merge.
- A “<” character indicates that the left text has been selected for merge.

It is also possible to edit the result line-by-line by selecting a line and edit it below the list. Clicking the *Change* button will replace the changed line in the result list.

Ignore moved or resized objects

If this option is on, the compare operation tries to ignore differences only caused by moved or resized symbols. You should only use this option if the two compared diagrams originates from the same diagram. (For instance, you have used Save as on the original diagram to get the second copy.) This option will not work well if you create two diagrams from scratch and compare them, because the operation uses object IDs to match moved or resized objects.

Select and Print Differences

If this option is on and there are differences, then a print dialog will pop up when the first dialog is closed. Only SDL diagram pages containing differences will be printed, and the differences will be highlighted with selection markers. If the same page containing differences can be found in both compared diagrams, then the two pages will be printed after each other, beginning with the page from the diagram that was visible in the editor when the compare operation was invoked.

Auto-merge using original file

This option is only available in the Merge Diagrams dialog. It allows you to specify the original diagram file that both the diagram versions that are compared originates from. If you specify the original diagram file, then the merge operation can auto-merge all non-conflicting changes for you. Note that if you give the wrong original diagram, then the auto-merge operation may merge the two versions in the wrong way. If you do not specify an original diagram, then the merge operation leaves all merging of differences to you.

Differences That Will Be Reported

When you use *Compare Diagrams* or *Merge Diagrams*, all visual object differences will be found:

- An object that has been moved, resized, removed, added or painted in a different color.
- A text, associated with an object, that has been changed

An object is considered to be moved if it has been moved relative to the upper left corner of the diagram frame. This means that if you move the frame, it will have no effect.

Diagrams are compared one page pair at a time. A page pair is one page from the original diagram and one page from the other diagram where both pages have the same name. If there is no page with the same name in the other diagram, then the complete page is considered to be a difference.

Difference Groups

Differing objects from the same page pair are grouped into *difference groups*. Two differing objects are placed in the same group if:

- The objects are from the same diagram and they are connected to each other. For instance, a flow line connected to a task symbol.
- The objects are from different diagrams and they would overlap if they were in the same diagram, that is, it would be impossible to place the two objects in the same diagram without changing the position within the diagram for one of them.
- The objects are from different diagrams and the difference operation can conclude that it is the same object that has been moved.

Color and the Current Difference Group

In the graphical difference report, all differing objects are given temporary background colors as specified in the initial dialog. As default, the color is green in the original diagram and magenta in the other diagram. Temporary colors are not saved in diagram files. How temporary colors are removed is described in "Temporary Colors > Remove" on page 2015.

The dialog that will be opened during the graphical difference report, allows you to navigate between difference groups using previous and next buttons. All objects in the current difference group are selected.

Normally, version 1 of the compared diagrams (the diagram that was in the SDL editor when the compare operation was invoked) is shown in a window in the top left corner of the screen. Version 2 (the diagram that was specified in the first compare dialog) is shown in the top right corner off the screen. For complete page differences (a difference where a page can not be found in the other diagram), the window that misses the page is hidden.

Note that it is possible to manually edit the compared diagrams while the compare operation is ongoing. This can be seen as a light-weight alternative to the full merge operation (described next).

Merging

During a merge operation, three windows are used in addition to the second merge dialog:

- The top left corner of the screen is used to show version 1 of the compared diagrams, in the editor window named *Source 1*. This is the diagram that was visible in the SDL editor window when the merge operation was invoked.
- The bottom left corner of the screen is used to show version 2 of the compared diagrams, in the editor window named *Source 2*. This is the diagram that was specified in the first merge dialog as the diagram to compare with.
- The right part of the screen is used to show the merge result diagram, in the editor window named *Result*. This is a new diagram that has been created from the two compared diagram versions.

The second Merge dialog does not only have previous and next buttons to navigate between differences. It does also have *Accept/Reject* buttons and “<<” (fast backward) / “>>” (fast forward) buttons.

- Pressing *Accept 1* (*2*) will add symbols to the merge result diagram. It is the symbols from version 1 (2) of the current difference group that will be added.
- Pressing *Reject 1* (*2*) will remove symbols from the merge result diagram. It is the symbols that can be found in version 1 (2) of the current difference group that will be removed.
- Pressing “<<” (fast backward) will navigate to the previous unresolved difference group. An unresolved difference group is a difference group where both version 1 and version 2 are not accepted.
- Pressing “>>” (fast forward) will navigate to the next unresolved difference group.

Note that it is possible to manually edit diagrams while a merge operation is ongoing. This can be useful if you want the merge result diagram to look different from both version 1 and version 2 of the compared diagrams.

When the second merge dialog is closed, the editor will ask you to save the merge result by displaying a save dialog above the window with the

merge result diagram buffer. When the save dialog is closed, the editor windows are restored to how they appeared before the merge operation was invoked.

GR to PR Conversion

It is possible to convert diagrams from SDL/GR to SDL/PR. In the conversion, SDL diagrams are translated from the binary files into PR files, possibly containing CIF information. All graphical symbols in that diagram will be replaced by their corresponding keywords and any text in the symbols will be copied to the output file. When possible, the definitions in the output file will appear in the same order as in the input.

The conversion is used by the SDL Analyzer as the first step in the analysis process, and also by the SDT2CIF Converter to convert to PR that preserve the graphical layout information. You can also convert to PR by the *Generate PR* command in the *Tools* menu, or by using the Public Interface (see "GRPR" on page 628 in chapter 11, *The Public Interface*).

The PR/CIF files produced by the SDL Suite fill various purposes:

- As a way to export information to other SDL tools
- As a temporary storage format that is used as input to the static and semantic analyzer
- As a format that is suitable for reading by the user. To produce this format, the PR file should be pretty printed.

Mapping between GR and PR

The following list defines the mapping between the GR and PR forms:

- In general, when symbols of the same type are converted, symbols are ordered first of all by the page ordering within the diagram. Within the same page symbols are ordered such that a symbol with a low y-coordinate (closer to the top) will be converted first. If two symbols have the same y-coordinate the symbol with the lowest x-coordinate is treated first.

Note:

The **order in which text symbols are converted may be of importance** when it comes to code generation issues. This is, in particular, applicable when using the SDL to C Compiler. Therefore, you should use one text symbol only for type definitions.

- When analyzing systems using macros, all macros are converted to PR before they are expanded. This restricts the usability of macros, for instance the number of inlets and outlets must be 0 or 1.
- A merge symbol (a flow line connected to another flow line) is translated to PR using join and label constructs. The labels introduced will be named *grst0*, *grst1*, and so on, starting on *grst0* at each diagram. In macro diagrams, the labeling is *grst0%MACROID*... If the label already exists in the GR diagram, the sequential number is incremental until the label is unique.
- A block diagram may contain a block substructure diagram without its frame and heading if the block diagram does not contain any definitions but the substructure. This rule is applied to all the pages in the block diagram that are of the block interaction type.
- When specifying a header in GR, it must be specified according to the PR syntax and the valid input signal set should be written into the heading symbol and not in a text symbol as specified by the GR syntax.
- References to graphical symbols, which will be used by the Analyzer in error messages, are stored in the PR file using SDL comments beginning with the character "#". It is therefore recommended that you do not write comments beginning with "#".
- To be able to perform a correct conversion of SDL/GR *Substructure* diagrams to SDL/PR, the enclosing diagram type, block/channel or block type, must be known. For a channel going to the environment, a name outside the frame must be a gate name, generating the `via` construct, if the enclosing diagram is a block type, and a channel name, generating `connect` statements, for other cases.

  If nothing is known about the enclosing diagram type, it is assumed that the diagram is a block.

  To resolve the ambiguity, a qualifier can be used in the kernel heading to determine the enclosing diagram type. Alternatively the An-

alyzer directive #BLOCKTYPE can be used to indicate that the enclosing diagram is a block type diagram. This directive should be added to the diagram kernel heading.

Differences when generating CIF

In general the PR generated as input to the analyzer will be the same as generated for CIF but for some constructs there are differences due to language limitations or to get a better error detection.

- The conversion of a graphical comment symbol differs dependent on if the translation is made to CIF or not. When converting to CIF, the PR generated is the `<comment>` PR construct; in other cases, including when performing Analysis, the graphical comment is converted to "/* text inside graphical comment symbol */".
- The SDL editor supports the possibility to place the comment symbol without connecting it to another symbol. When generating PR without CIF this comment will be converted to PR as described above. When generating CIF the CIF comment will be
  /* CIF Comment (500,100) Right */
  /* text in unconnected comment */
  This standalone nonstandard CIF construct will be recognized by the CIF2SDT application such that the original comment symbol will be restored in the diagram.
- In the SDL editor you can connect many comments to the same symbol. This is an extension to the SDL language. For this graphical construct there is no immediate translation to PR. When generating PR without CIF the comments are generated according to the note above and there will be no problems. However, when generating CIF the comment will be generated using the <comment> PR construct. As there only can be one <comment> to each symbol there is no direct possible translation in PR for many comments. In this case the extra comment symbols are handled as if they were unconnected to the symbol. This means that when regenerating a diagram from the CIF code, the diagram will not be exactly the same as the original diagram as the extra comments will now be unconnected.
- When generating non-CIF all unattached comments will be attached to a symbol that is closest to the comment. In this way the comment text will be generated as if the text belongs to this symbol.

- There is a difference also for graphical connections. If a channel C1 has a graphical connector to the outer channel outerC and another channel C2 also has a graphical connector to outerC, but the points where outerC is drawn is not the same, the conversion for CIF will contain two connect statements:

  ```
  /* CIF ... */
  connect outerC and C1;
  /* CIF ... */
  connect outerC and C2;
  ```

  This PR will produce a semantic error according to SDL-92, but will be correct when using SDL-96.

  When using non-CIF conversion, the two graphical connections will be combined into one statement, provided that the two texts specifying outerC are lexically the same, thus writing outerC in one text and out_\nerC in the other (\n is a newline) will not be treated as the same text. They are still considered the same if only the case differs.

  ```
  connect outerC and C1,C2;
  ```

- When PR is generated for a macrodefinition the macro inlet and macro outlet symbols will differ whether CIF is generated or not. As there are no corresponding PR to these symbols nothing will be generated when generating CIF except of course for the CIF statements itself. For non-CIF a note will generated indicating that these symbols are present in the diagram.

  ```
  /* Macro Inlet Symbol  */
  ```

  A task having no text or having only a note/comment is allowed in GR. The correct PR correspondence is the compound statement construct having an empty statement

  ```
  task { };
  ```

  As this PR will restrict the use of the simulator command Next-Symbol the PR generated for non-CIF is an empty informal text

  ```
  task '';
  ```

  For CIF the compound statement will be generated.

- CIF generation for the class symbol is not supported. The generation for non-CIF is described in "Class Information" on page 1942.

Error and Warning Message

This section contains messages that may be produced during the GR to PR conversion.

When the Public Interface performs the conversion from GR to PR, the messages will be displayed in the Organizer log. The reply message on the service will indicate that the conversion has been done and it will also contain the number of errors produced.

If the conversion for some reason fails, the reply message will contain a text stating the kind of problem that has occurred. See further the *GRPR* service in the Public Interface.

ERROR 5001 Unexpected end of flow

This message indicates that a symbol has been found that has no following flowline and according to the SDL syntax there must be one. In addition to the error message the following line is generated in the PR:

```
***** Unexpected end of flow *****
```

WARNING 5002 More than one line entering macro outlet

This message indicates that inside a macro there are more than one line entering the macro outlet symbol. This situation should be avoided as this can lead to syntax errors (which are hard to track) if this macro is expanded inside decision branches.

The easiest way to avoid this situation is to add a no-action task symbol with the text ‘’ and use this as the last symbol in the macro. Let all lines enter this task symbol and add a single line between this task and the macro outlet.

Chapter 43 Using the SDL Editor

Chapter

44

Symbols and Lines – Quick Reference

The following notations are included in the quick reference:

- **DP - Deployment Diagrams**
- **HMSC – High-level Message Sequence Charts**
- **MSC – Message Sequence Charts**
- **OM – Object Model**
- **SC – State Charts**
- **SDL – Specification and Description Language**

The quick reference contains all the symbols and lines in those diagrams in combination with a short explanation of when they may be used.

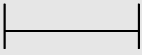
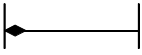
There is also a landscape oriented version of this document. It is called `symref.pdf` and is located in `/pdf/files` on the installation CD. If you want to make your own printout of the quick reference, that document is perhaps the most suitable.

Symbols and Lines in DP Diagrams

Symbols in DP Diagrams

Symbol	Name	Explanation
	Node	Is used to describe a run-time physical object that represents a computational resource.
	Component	Represents a physical, replacable part of a system that packages implementation.
	Thread	Represents an OS thread.
	Object	Represents an SDL entity (system, block or process).
	Text	Contains comments relevant for the diagram or build script information. Is not connected to any other symbol.

Lines in DP Diagrams

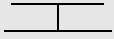
Line	Name	Explanation
	Association line	Defines an association between two node symbols. To create it, select one node symbol and drag the association line handle (a rectangle). To create a line breakpoint, click in the diagram background. To attach the association line to its final destination, click the border of the associated node symbol.
	Composite Aggregation line	Defines that the symbol connected to the simple line end is contained in the symbol connected to the diamond line end. To create it, select the container symbol and drag the aggregation line handle (a diamond). To create a line breakpoint, click in the diagram background. To attach the aggregation line to its final destination, click the border of the contained symbol.

Symbols and Lines in HMSC Diagrams

Symbols in HMSC Diagrams

Symbol	Name	Explanation
	Start	The start symbol in an HMSC diagram. Is followed by a condition, reference or a condition point symbol.
	Stop	The last symbol in an HMSC diagram.
	Condition	Represents a system or process state, or indicates that a certain condition is true. Contains the name of the represented state or condition.
	Reference	References another MSC or HMSC diagram in the same group of MSC diagrams. Contains the name of the other diagram.
	Connection point	Splits or joins lines. One to many, many to one, or many to many.
	Text	Contains comments relevant for the diagram. Is not connected to any other symbol.

Lines in HMSC Diagrams

Line	Name	Explanation
	HMSC line	Defines how symbols can be traversed. Starts below the from-symbol and ends above the to-symbol. From-symbols can be start, condition, reference and connection point. To-symbols can be stop, condition, reference and connection point. To create the line, select a start, condition, reference or connection point symbol (a from symbol), and drag the line handle. To create a line endpoint, click in the diagram background. To attach the line to its final destination, click the symbol border.

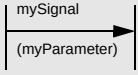
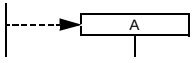
Symbols and Lines in MSC Diagrams

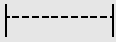
Symbols in MSC Diagrams

Symbol	Name	Explanation
	Text	Contains comments relevant for the diagram. Is not connected to any other symbol.
	Comment	Contains comments relevant for a part of the diagram. Is connected to a symbol, a message sending or a message reception.
	Instance head	Represents an instance of something that can communicate by sending and receiving messages. The “start symbol” for a created or an already existing instance.
	Instance end	Used for graphically ending an instance axis without terminating the represented instance.
	Message	Represents a signal sending from one instance to another. Is connected to at least one instance axis. Usually connected to an instance axis in both ends.
	Condition	Represents a system or process state, or indicates that a certain condition is true. Contains a name of the represented state or condition. Is initially connected to one instance axis, but should often be connected to all.
	Timer	Represents the use of a timer in an instance. Is connected to an instance axis. One symbol represents both a timer set and either a timeout or a reset.
	Action	Is a kind of text symbol and is connected to an instance axis. Describes something that is happening in the instance.
	Create process	Describes the creation of an instance. The arrow should go from an instance axis (the creator) to an instance head (the creator instance). The instance head is created when the create process symbol is initially laid out.
	Process stop	Terminates an instance. The instance ceases to exist. Is connected to an instance axis.

Symbol	Name	Explanation
	Coregion	Creates a part of an instance axis where the order of received and sent signals is undefined. Is connected to an instance axis.
	MSC reference	References another MSC or HMSC diagram in the same group of MSC diagrams. Contains the name of the other diagram. Is initially connected to one instance axis, but can be connected to several.
	Inline expression	Is used for specification of alternative or optional parts (that is, message sending sequences) which are contained in the symbol. Is initially connected to one instance axis, but should most likely be connected to several.
	Inline expression separator	Creates a new partition in an inline expression symbol. Initially, the inline expression symbol only contains one partition.

Lines in MSC Diagrams

Line	Name	Explanation
	Message	Is used for defining a message sending and/or a message reception. Is at least connected to one instance axis, but is usually connected to an instance axis in both ends. Create it from the symbol menu.
	Create process line	Is used for defining the creation of an instance. The arrow should go from an instance axis (the creator) to an instance head (the created instance). Create the process line from the symbol menu. When the create process arrow is initially laid out, the instance head is created.

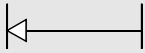
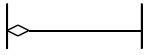
Line	Name	Explanation
	Comment line	Associates a comment symbol with another symbol. To create it, select the comment symbol and drag the line handle to another symbol. Note that it is also possible to attach the comment symbol to a message sending or a message reception, by dragging the line handle to a connection point between a message and an instance axis.

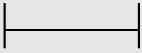
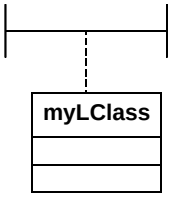
Symbols and Lines in OM Diagrams

Symbols in OM Diagrams

Symbol	Name	Explanation
	Class	Specifies a class. Contains three sections: Name, attributes and operations.
	Object	Specifies an object instantiated from a class. Contains two sections: Name and attribute values
	Text	Contains comments relevant for the diagram. Not connected to any other symbol.

Lines in OM Diagrams

Line	Name	Explanation
	Generalization line	Defines that the class connected to the simple line end inherits from the class connected to the triangle line end. To create it, select the class symbol to inherit from and drag the generalization line handle (a triangle). To create a line break point, click in the diagram background. To attach the generalization line to its final destination, click the border of the inheriting class symbol.
	Aggregation line	Defines that the class connected to the simple line end is contained in the class connected to the diamond line end. To create it, select the container class symbol and drag the aggregation line handle (a diamond). To create a line breakpoint, click in the diagram background. To attach the aggregation line to its final destination, click the border of the contained class symbol.

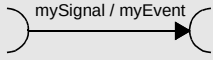
Line	Name	Explanation
	Association line	Defines an association between two class symbols. To create it, select one class symbol and drag the association line handle (a rectangle). To create a line breakpoint, click in the diagram background. To attach the association line to its final destination, click the border of the associated class symbol.
	Link class line	Defines a class connected to an association line or an aggregation line. To create it, select an association or an aggregation line and drag the link class handle. The class symbol is created at the same time. There can only be one class symbol connected to one association or aggregation line.

Symbols and Lines in SC Diagrams

Symbols in SC Diagrams

Symbol	Name	Explanation
	State	Represents a state in a state machine. May contain sub states when hierarchical state machines are specified. Is connected to other states via transition lines.
	Start	The start symbol on a level in a state chart. Is followed by a transition to a state symbol.
	Termination	Terminates a state machine. The state machine ceases to exist.
	Text	Contains comments relevant for the diagram. Is not connected to any other symbol.

Lines in SC Diagrams

Line	Name	Explanation
	Transition line	Defines a transition between two states. Is also used for defining the start transition and the last transition, by connecting a state symbol with either a start symbol or a termination symbol. To create it, select the start symbol or a state symbol and drag the transition line handle. To create a line breakpoint, click in the diagram background. To attach the transition line to its final destination, click the border of a state or termination symbol.

Symbols and Lines in SDL Diagrams

Symbols in Both SDL Structure and Behavior Diagrams

Symbol	Name	Explanation
	Class	Defines a newtype. Contains three sections: Name, attributes and operators.

Symbols in SDL Structure Diagrams

Symbol	Name	Explanation
	Text	Declares SDL entities such as signals and data types.
	Comment	Contains comments in the diagram. Is attached to other symbols.
	Text extension	Is attached to other symbols. Used if the text in another symbol is too large for the symbol. Put the last part of the text or the complete text in the text extension symbol.
	Block reference	References a block diagram in a system, system type, substructure, block or block type diagram. Also used for instantiation of a block type.
	Process reference	References a process diagram in a block or block type diagram. Is also used for instantiation of a process type
	Block substructure reference	References a substructure diagram from a block or block type diagram.[a]
	Service reference	References a service diagram in a process or process type diagram, from a service interaction page. Also used for instantiation of a service type.

Symbol	Name	Explanation
	Procedure reference	References a procedure diagram from any other diagram.
	System type reference	References a system type diagram from a package diagram.
	Block type reference	References a block type diagram from a package, system, system type, block or block type diagram.
	Process type reference	References a process type diagram from a package, system, system type, block or block type diagram.
	Service type reference	References a service type diagram from a package, system, system type, block, block type, process or process type diagram.
	Operator reference	References an operator diagram from any other diagram.
	Gate	Defines a gate in a block type, process type or service type diagram. Is attached to the diagram frame.

a. Not often used, because “block A in block C” is a commonly used shorthand for the more syntactically correct “block A in substructure B in block C”.

Symbols in SDL Behavior Diagrams 1(2)

Symbol	Name	Explanation
	State	Defines a state or terminates a transition in an already defined state. Is followed by an input, save continuous signal or priority output signal.
	Input	Receives a signal. Always preceded by a state symbol. Together, they define the start of a transition. Is followed by the behavior of the transition.
	Save	Saves signals from being discarded when being received in the current state (that does not handle the signal). Always preceded by a state symbol. Is not followed by any symbols.

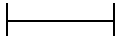
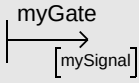
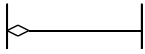
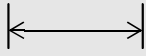
Symbol	Name	Explanation
	Output	Sends a signal from a transition
	In/out connector	**Out connector:** A jump/join/goto symbol that finishes the definition of a transition on one page, if there is not enough space. Is always associated via a name with an in connector that continues the definition. **In connector:** The label symbol that is followed by the rest of the transition.
	Procedure call	Calls a procedure that does not return a value from a transition. (A value returning procedure is called from a task symbol.)
	Create request	Creates an instance of a process in a transition
	Enabling condition/ continuous signal	**Enabling condition:** Is preceded by an input symbol. Contains a boolean expression that decides if the transition below it should be taken or not. **Continuous signal:** Is preceded by a state symbol. Is followed by the behavior of a transition. Contains a boolean expression that is continuously evaluated while in the state. The following transition is taken when the expression evaluates to true.
	Priority input	Specifies that this signal reception has higher priority than normal signal reception in the same state. Primarily intended to give signals between services in the same process higher priority than other signals. Not often used.
	Procedure start	The start symbol in a procedure diagram. Is followed by the behavior of the start transition.
	Procedure return	The symbol in a procedure diagram that finishes the execution of the procedure and returns to the procedure caller.
	Procedure reference	References a procedure diagram from any other diagram.
	Gate	Defines a gate in a block type, process type or service type diagram. Is attached to the diagram frame.

Symbols in SDL Behavior Diagrams 2(2)

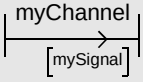
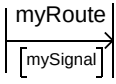
Symbol	Name	Explanation
	Text	Declares the SDL entities such as variables, timers and types.
	Comment	Contains comments in the diagram. Is attached to other symbols.
	Text extension	Is attached to other symbols. Used if the text in a symbol is too large for the symbol. Put the last part of the text or the complete text in the text extension symbol.
	Decision	Specifies alternative paths in the behavior part of a transition. Contains an expression. Each path is labeled with an answer that should match the expression for the path to be taken.
	Task	Is used for writing textual code in the behavior part of a transition. Contains for example variable assignments, for-loops and calls of value returning procedures.
	Macro call	Calls a macro diagram in the behavior part of a transition. Macros are a depreciated feature of SDL, consider using procedures or diagram inheritance instead.
	Transition option	Specifies alternative paths in the behavior part of a transition. Similar to a decision symbol, but the expression must be built up of constants and (external) synonyms that can be evaluated once before execution of the system starts. Not often used.
	Start	Starts the execution of the current diagram instance in a process, process type, service or service type diagram. Is followed by the definition of the behavior of the start transition.
	Stop	Stops the execution of the current diagram instance in a process, process type, service or service type diagram.
	Macro inlet	The start symbol in a macro diagram. Macros are a depreciated feature of SDL, consider using procedures or diagram inheritance instead.

Symbol	Name	Explanation
	Macro outlet	The end symbol in a macro diagram. Macros are a depreciated feature of SDL, consider using procedures or diagram inheritance instead.
	Operator reference	References an operator diagram from any other diagram.

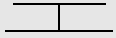
Lines in Both SDL Structure and Behavior Diagrams

Line	Name	Explanation
	Comment line	Associates a comment symbol with another symbol. To create it, select the comment symbol and drag the line handle to another symbol.
	Text extension line	Associates a text extension symbol with another symbol. To create it, select the text extension symbol and drag the line handle to another symbol.
	Gate line	Defines a gate in a block type, process type or service type diagram. To create it, select it from the symbol menu and attach it to the diagram frame.
	Aggregation line	Defines that the class connected to the simple line end is contained in the class connected to the diamond line end. To create it, select the container class symbol and drag the aggregation line handle (a diamond). To create a line breakpoint, click in the diagram background. To attach the aggregation line to its final destination, click the border of the contained class symbol.
	Association line	Defines an association between two class symbols, can be unidirected or bidirected. To create it, select one class symbol and drag the association line handle (a rectangle). To create a line breakpoint, click in the diagram background. To attach the association line to its final destination, click the border of the associated class symbol.

Lines in SDL Structure Diagrams Only

Line	Name	Explanation
	Channel	Defines a part of a communication path for signals, either between two blocks or between one block and the environment (i.e. the diagram frame). To create it, select a block symbol and drag the line handle. To make line breakpoints, click in the diagram background. To attach the line to its final destination (either another block or the diagram frame), click the symbol border. After creation, the channel can be reversed or made bidirectional. The arrow is movable and an arrow at the end of the channel indicates that the channel does not delay signals being sent.
	Signal route	Similar to a channel. The differences are that a signal route connects to processes and services instead of blocks, and that a signal route arrow cannot be moved around.
	Create line	Defines that the process where the create line starts, can create instances of the process where the create line ends. The create line is optional. To create it, select the process symbol and drag the create line handle. To make line breakpoints, click in the diagram background. To attach the create line to its final destination, click the symbol border.

Line in SDL Behavior Diagrams Only

Line	Name	Explanation
	Flow line	Defines the order that symbols are executed in. Starts at the bottom of a symbol and ends at the top of another symbol. To create it, select a symbol that can be followed by other symbols and drag the flow line handle. To make line breakpoints, click in the diagram background. To attach the flow line to its final destination, click the symbol border. Note that two symbols can only be connected with a flow line if the SDL syntax rules allow it.

Chapter

45 *The SDL Type Viewer*

The Type Viewer shows SDL-92 types and type instances in the current system and how they relate to each other.

This chapter contains a reference manual to the Type Viewer; the functionality it provides, its menus, windows and symbols.

Objects and Windows

Objects and Attributes

An object in the Type Viewer is either a type or an instance of a type. Both of these may be referred to as a *type* in the Type Viewer and in this chapter. For each type object, the following attributes can be presented:

- A graphical *symbol* to identify the type of diagram. The symbols are identical to the icons used in the Organizer's Main window (see "Icon Types" on page 51 in chapter 2, *The Organizer*). The symbol is always shown.
- The *type identifier*, in the form <virtuality> <diagram type> <object type>, where:
  - <virtuality> may be either Virtual, Redefined or Finalized
  - <diagram type> may be either System, Block, Service or Process
  - <object type> may be either Type or Instance
- The *name* of the type or instance
- The complete *qualifier*, in text form.

You can hide or show each of the last three text items. By default, the type identifier and the type name is shown.

You can open an SDL Editor by double clicking on a symbol in the Type Viewer. If a type symbol is double clicked, an SDL Editor is opened with that type diagram. If an instance symbol is double clicked, an SDL Editor is opened with the diagram that contains the instance.

Type Viewer Windows

The Type Viewer has two windows, the list window and the Tree window, which present the type information in the system in two different ways. The main window lists the type objects without showing inheritance or redefinition relations between them, whereas the Tree window shows such relations for a type object in a tree form. Both windows can

display all of the object attributes above, but in somewhat different ways.

The Main window is opened when the Type Viewer is started, whereas the Tree window is not opened initially. The general characteristics of these windows are described in chapter 1, *User Interface and Basic Operations*.

Updating the Type Viewer

When the Type Viewer is started, it extracts type information from the current system. If, for any reason, there are problems extracting type information, you are warned in a dialog and details can be found in the Organizer log window. You may update the information in the Type Viewer at any time, with the *Update* command.

The Type Viewer obtains the information about what types are defined in the SDL scope from the Information Server. The Information Server updates its contents **each time an SDL diagram is saved**. When the diagram is saved, the Analyzer is invoked and produces a mirrored image which is an SDL/PR description of the SDL/GR diagram. That information is then loaded into the Information Server which serves the Type Viewer with information upon request. All of this is done automatically without the user having to bother about performing any actions.

Error Reporting

> **Note:**
>
> In order to extract signal information from a diagram, the diagram needs to be syntactically correct in the sense that SDL/PR code can be generated from the SDL/GR diagram without encountering errors.

Any errors are appended to the Organizer log window. For a reference to error messages that may be produced by the Information Server, see *"Information Server Error Messages" on page 912 in chapter 17, The Information Server*.

Main Window

The Drawing Area

The main window's drawing area contains a simple line-oriented list of all types in the current SDL system. No inheritance or redefinition relations are shown. The object attributes are listed in the following way (see Figure 443):

Icon Type identifier Type name
<< Qualifier >>

The qualifier is displayed within double angle brackets on a single text line, with slashes separating the path items.

In the list, the type objects are sorted alphabetically according to the diagram type, object type and type name, i.e. all objects of the same diagram type are grouped together.

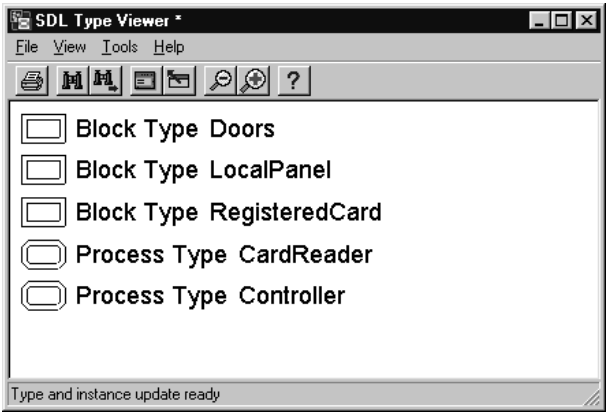


Figure 443: Main window

When you select a type or instance in the main window, the Tree window is updated and displays the relation trees that contain the selected type/instance. If the Tree window has not been opened earlier in the same Type Viewer session, the Tree window is opened automatically.

The Menu Bar

File Menu

The *File* menu contains the following menu choices:

- *Update*
- *Print*
  (See *"Printing from the SDL Suite" on page 315 in chapter 5, Printing Documents and Diagrams*.)
- *Exit*
  (See "Exit" on page 15 in chapter 1, *User Interface and Basic Operations*.)

Update

Updates the type list in the main window and the type trees in the Tree window, according to the possibly changed SDL diagrams (see "Updating the Type Viewer" on page 2071).

This command should be used whenever changes that might have an impact on the type inheritance are applied to any of the SDL diagrams contained in the diagram structure.

View Menu

The View menu contains the following menu choices:

- *Window Options*
- *List Options*
- *Symbol Options*
- *Set Scale*

Window Options

Sets options for which parts of the main window to show. The dialog controls whether to show the tool bar and the status bar.

List Options

Sets options for which symbols to include in the list in the main window. The dialog controls whether to show *Instance symbols* (type symbols are always shown).

Symbol Options

Sets options for which attributes to show for each type object in the list in the main window. The dialog controls whether to show the *Qualifier*, the *Type* identifier, and the *Name* of the types and instances.

Set Scale

Issues a dialog where the scale for the main window may be set.

Tools Menu

The Tools menu contains the following menu choices:

- *Show Organizer*
  (See "Show Organizer" on page 15 in chapter 1, *User Interface and Basic Operations*.)

- *Search*
- *Search Again*
- *Show in Editor*
- *Show Type Trees*

Search

Searches for a visible text in the type list shown in the Main window.

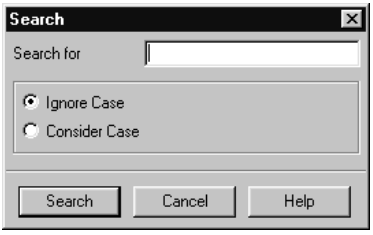


Figure 444: The Search dialog

- *Search for*

  Specifies the text string to search for. As a special case, a GR Reference obtained from for instance an editor may also be specified.

- *Ignore Case, Consider Case*

These options toggles between a case insensitive and a case sensitive search.

- *Search*

  Starts the search and closes the dialog. The first symbol containing the text is selected. The search starts from the selected symbol, if any, or from the first symbol in the list.

If the search reaches the end of the list, you are asked whether to continue the search from the top of the list or not.

If the text cannot be found among the visible information, you are informed in a confirmation dialog.

Search Again

Searches again for the same text as in the latest search. The behavior is the same as described in "Search" on page 2074.

Show in Editor

Shows the selected type or instance in an SDL Editor. If a type symbol is selected, an SDL Editor is opened with that type diagram. If an instance symbol is selected, an SDL Editor is opened with the diagram that contains the instance.

Show Type Trees

Opens or raises the Tree window. The type or instance selected in the Main window becomes selected in the Tree window.

Popup Menus

The following tables lists the menu choices in the Main window popup menus and a reference to the corresponding menu choice in the menu bar.

On the Main Window Background

Search	“Search” on page 2074.
Search Again	“Search Again” on page 2075.
Show Type Trees	“Show Type Trees” on page 2075.

On a Symbol in the Main Window

Search	“Search” on page 2074.
Search Again	“Search Again” on page 2075.
Show in Editor	“Show in Editor” on page 2075.
Show Type Trees	“Show Type Trees” on page 2075.

Keyboard Accelerators

Apart from the general keyboard accelerators, the following accelerator can be used in the main window:

Accelerator	Reference to corresponding command
`Ctrl+E`	“Show in Editor” on page 2075

Tree Window

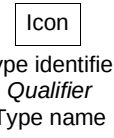
The Drawing Area

The Tree window's drawing area visualizes inheritance and redefinition relations between types by presenting the types and instances in a tree form. There are two types of trees displayed in the Tree window:

- A *Redefinition tree* shows virtual types, specializations based on redefinition of the virtual types, and all instances of these types.
- An *Inheritance tree* shows types, specializations based on inheritance and adding properties to the types, and all instances of these types.

The two trees for a certain type are often, but not always, identical. The Type Viewer supports you by checking if they are identical; if so, they are shown as one tree. When the Inheritance and Redefinition trees are different, both are shown separately. Each tree has a header identifying which type of tree it is.

The object attributes are listed vertically in the trees in the following way (see Figure 445):



The qualifier is displayed in italics in a multi-line form, with each path item on a separate line. No angle brackets or slashes are used.

In the tree, the type objects with the same parent are placed from left to right alphabetically according to the diagram type and object type.

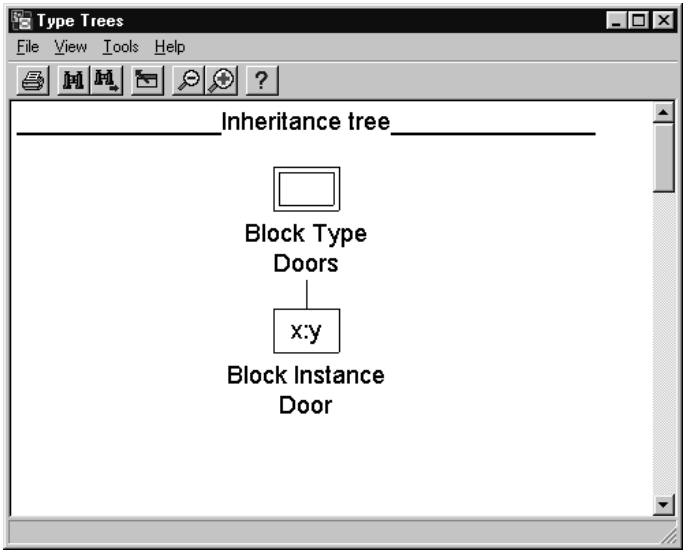


Figure 445: Tree window

The Tree window is updated when the selection in the Main window is changed. The Tree window is updated to show the Inheritance and Redefinition trees in the system that contain the type/instance that is selected in the Main window. The first occurrence of that type/instance also becomes selected in the Tree window.

It is possible to view all Inheritance and Redefinition trees in the current system at once in the Tree window (see "Show All Trees" on page 2080). The trees are lined up vertically in the same drawing area.

The Menu Bar

File Menu

The *File* menu contains the following menu choices:

- *Print*
  (See *"Printing from the SDL Suite" on page 315 in chapter 5, Printing Documents and Diagrams*.)
- *Close*
  (See "Close" on page 14 in chapter 1, *User Interface and Basic Operations*.)

View Menu

The View menu contains the following menu choices:

- *Expand*
- *Expand Substructure*
- *Expand All*
- *Collapse*
- *Window Options*
- *Tree Options*
- *Symbol Options*
- *Show All Trees*
- *Set Scale*

Expand

Expands the selected object one level down.

Expand Substructure

Expands the selected object as much as possible.

Expand All

Expands all visible trees as much as possible.

Collapse

Collapses the selected object, i.e. hides all child objects. If there is no selected object, the menu choice is named *Collapse All*. *Collapse All* collapses all trees in the window.

Window Options

Sets options for which parts of the Tree window to show. The dialog controls whether to show the tool bar and the status bar.

Tree Options

Sets options for which symbols to include in the trees in the Tree window. The dialog controls whether to show instance symbols (Type symbols are always shown).

Symbol Options

Sets options for which attributes to show for each symbol in the trees in the Tree window. The dialog controls whether to show the *Qualifier*, the *Type* identifier, and the *Name* of the types and instances.

Show All Trees

Shows type trees for all types in the system.

Set Scale

Issues a dialog where the scale for the Tree window may be set.

Tools Menu

The Tools menu contains the following menu choices:

- *Show Organizer*
  (See "Show Organizer" on page 15 in chapter 1, *User Interface and Basic Operations*.)

- *Search*
- *Search Again*
- *Show in Editor*
- *Show Type Viewer*

Search

Searches for a text in the type trees shown in the Tree window.

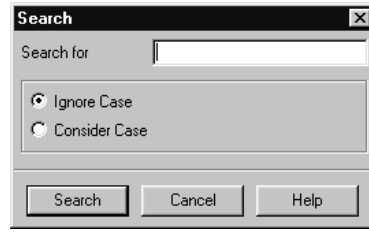


Figure 446: The Search dialog

- *Search for*

  Specifies the text string to search for. As a special case, a GR Reference obtained from for instance an editor may also be specified.

- *Ignore Case, Consider Case*

  These options toggles between a case insensitive and a case sensitive search.

- *Search*

  Starts the search and closes the dialog. The first symbol containing the text is selected. The search starts from the selected symbol, if any, or from the first symbol in the first tree.

If the search reaches the end of the trees, you are asked whether to continue the search from the first tree.

If the text cannot be found, you are informed in a confirmation dialog.

Search Again

Searches again for the same text as in the latest search. The behavior is the same as described in "Search" on page 2080.

Show in Editor

Shows the selected type or instance in an SDL Editor. If a type symbol is selected, an SDL Editor is opened with that type diagram. If an instance symbol is selected, an SDL Editor is opened with the diagram that contains the instance.

Show Type Viewer

Raises the Type Viewer's Main window.

Popup Menus

The following tables lists the menu choices in the Tree window popup menus and a reference to the corresponding menu choice in the menu bar.

On the Tree Window Background

Search	"Search" on page 2080.
Search Again	"Search Again" on page 2081.
Show All Trees	"Show All Trees" on page 2080.
Show Type Viewer	"Show Type Viewer" on page 2082.
Expand All	"Expand All" on page 2079.
Collapse All	Collapse all top objects in all trees.

On a Symbol in the Tree Window

Search	"Search" on page 2080.
Search Again	"Search Again" on page 2081.
Show in Editor	"Show in Editor" on page 2081.
Show All Trees	"Show All Trees" on page 2080.
Show Type Viewer	"Show Type Viewer" on page 2082.
Expand	"Expand" on page 2079.
Expand Substructure	"Expand Substructure" on page 2079.
Collapse	"Collapse" on page 2079.

Keyboard Accelerators

Apart from the general keyboard accelerators, the following accelerator can be used in the Tree window:

Accelerator	Reference to corresponding command
Ctrl+E	“Show in Editor” on page 2081

Chapter 45 The SDL Type Viewer

Chapter

46 *The SDL Index Viewer*

The Index Viewer shows all definitions of SDL entities in a system and where they are used. The Index Viewer uses cross reference files produced by the Analyzer.

In previous versions of SDL Suite, the Message Sequence Chart Editor could also produce listings of definitions and references from an MSC. These older listings are also supported by the Index Viewer.

This chapter contains a reference manual to the Index Viewer; the functionality it provides, its menus, windows and symbols.

Entities and Windows

Definitions and Uses

SDL

The SDL Analyzer has the ability to generate files containing SDL cross references. See the Analyzer command described in *"Set-Xref" on page 2496 in chapter 54, The SDL Analyzer*, and the Organizer's Analyzer option described in "Generate a cross reference file" on page 116 in chapter 2, *The Organizer*. These files use a textual format that is human readable; the purpose of the Index Viewer is to provide a means to take advantage of this information in the graphical environment. SDL icons are used when displaying the contents of the files in a graphical way.

MSC

In previous versions, the Message Sequence Chart Editor featured the generation of MSC cross references. These entities had properties that were similar to SDL cross references. Where possible, a direct mapping from MSC to the corresponding SDL concept were made. A few concepts were added so that all information of interest could be extracted from an MSC.

MSC icons identify the various MSC entities graphically, and are supported for backward compatibility reasons.

SDL Icons

Each type of SDL entity has a unique name and icon used in the Index Viewer. The different SDL diagram types use the same icon and type name as in the Organizer's Main window. For other types of entities, the following icons and names are used:

Icon	Name
	OPERATOR
	PACKAGE_INTERFACE
	CHANNEL

Icon	Name
	SIGNALROUTE
	GATE
	INPUT
	OUTPUT
	CONNECTION
	CREATE
	TASK
	SIGNAL
	SIGNALLIST
	GENERATOR
	FPAR; PROCESS_PARAM, PROCEDURE_PARAM (actual param.)
	STATE
	NEXTSTATE
	TIMER
	LABEL (in-connector, out-connector)
	INSTANTIATION
	SET
	USE
	INHERITS
	JOIN
	STATE_LIST
	ATLEAST
	DECISION

Icon	Name
	SAVE
	DCL; VARIABLE
	NEWTYPE; SYNTYPE; SORT
	SYNONYM
	RESET
	IMPORT; IMPORT_DEF (i.e. Imported)
	LITERAL
	VIEW; VIEW_DEF (i.e. Viewed)
	REMOTE_PROCEDURE
	REMOTE_VARIABLE
	SIGNALSET
	ACTIVE
	TRANS_OPTION (i.e. transition option)
	EXPORT
	ENAB_COND (i.e. enabling condition)
	CONT_SIGNAL (i.e. continuous signal)
	NUMBER_INST (i.e. number of instances)
	EXPORTED_PROCEDURE
	(PROCEDURE) CALL
	REFERENCE
	FIELD
	Unknown symbol type. Used for diagrams when the type of the diagram for some reason could not be determined.

MSC Icons

Each type of MSC entity has a unique name and icon used in the Index Viewer. The MSC diagram uses the same icon as in the Organizer's Main window. For other types of entities, the following icons and names are used:

Icon	Name
	CONDITION
	CREATE of instance
	INPUT of message
	INSTANCE head
	OUTPUT of message
	RESET of timer
	SET of timer
	STOP of process
	TIMEOUT (expire of timer)
	Unknown symbol type. Used for diagrams when the type of the diagram for some reason could not be determined.

Index Viewer Window

All entities defined in the system are listed in alphabetical order in the Index Viewer window. The entities are sorted alphabetically according to either name, or type and name. For each entity, there may be a list of uses of the entity. The general characteristics of the window is described in chapter 1, *User Interface and Basic Operations*.

The Drawing Area

The window contains a list of all definitions of entities in the current SDL system or the current MSC diagram. The first line in the list gives the name of the current cross reference file.

For each entity, the following information is presented:

- An icon identifying the type of entity (see "SDL Icons" on page 2086 and "MSC Icons" on page 2089)
- The name and type of the entity.
- A reference to the diagram the entity definition is located in.
- The number of definitions with an asterisk in front, if more than one definition of the entity exists (in SDL diagrams, it is only states that can have multiple places of definition for one entity).
- The number of cross references found (only for SDL entities; not applicable to MSC entities)

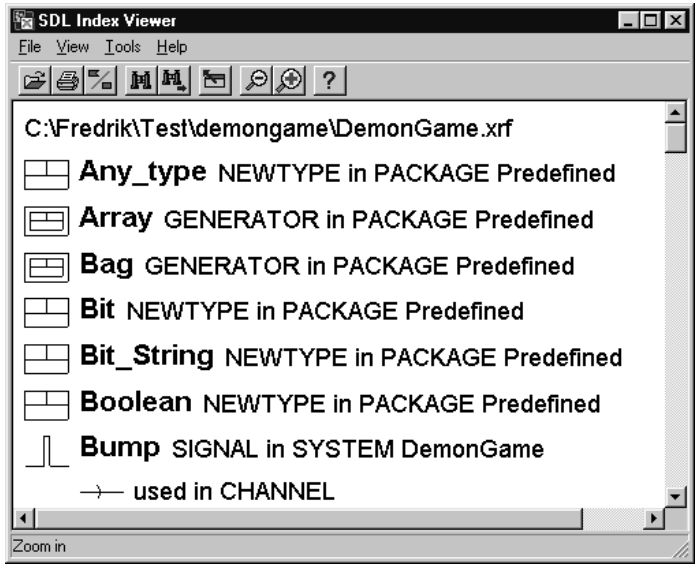


Figure 447: The Index Viewer window

To double click on a defined entity in the window is the same operation as *Show Definition* in the *Tools* menu (see "Show Definition" on page 2097). If there are several definitions associated with the entity, one of them is selected. If you double click again, another definition is selected. The list of definitions associated with the entity is traversed in a circular pattern.

To double click on a use in the window is the same operation as *Show Use* in the *Tools* menu (see "Show Use" on page 2098). If there are several uses associated with the symbol, one of them is selected. If you double click again, another use is selected. The list of uses associated with the reference symbol is traversed in a circular pattern.

Fast Search

There is a fast search operation available when the Index Viewer drawing area has the input focus. Just start typing on the keyboard, and the list of entities will be searched from the top. The first matching entity will be selected.

Note that the fast search operation will only match if what you type matches the beginning of the text for an entity. For instance: If you type “Ga”, you will find the entity *Game_On* if the entities are sorted with name first, while you will not find it if you type “On”. The normal *Search* operation does not have this restriction.

The search string used by the fast search operation will be reset whenever any other operation than the fast search operation is performed. For instance, if you have typed “Ga” and the index viewer has found the entity *Game_On*, and you select another entity and start typing again, then the search string used for the search operation will not contain the initial “Ga” characters (and the fast search operation will start searching from the top of the list).

It is possible to use the delete character to correct any spelling mistakes. You can see what you have typed in the message area. Matched characters are shown in non-capital letters, while unmatched are shown in capital letters.

The Menu Bar

File Menu

The *File* menu contains the following menu choices:

- *Open*
  (See “Open” on page 9 in chapter 1, *User Interface and Basic Operations*.)
- *Print*
  (See *“Printing from the SDL Suite” on page 315 in chapter 5, Printing Documents and Diagrams*.)
- *Exit*
  (See “Exit” on page 15 in chapter 1, *User Interface and Basic Operations*.)

View Menu

The *View* menu contains the following menu choices:

- *Window Options*
- *Index Options*
- *Filter Types*

- *Filter Uses*
- *Filter Diagrams*
- *Set Scale*

Window Options

Sets options for which parts of the Main window to show. The dialog controls whether to show the tool bar and the status bar.

Index Options

Determines the appearance of each entity in the window.

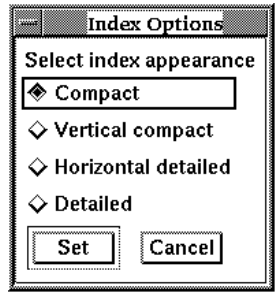


Figure 448: The Index Options dialog

The following alternatives are available:

- Compact:

  T timer in process Demon 3 uses

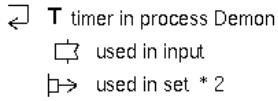
- Vertical Compact:

  T timer in process Demon
  3 uses

- Horizontal Detailed:

  T timer in process Demon used in input used in set * 2

- Detailed:



Filter Types

Determines which entity types to show/hide in the entity list. A dialog is opened with a sorted list of all entity types found in the current cross reference file:

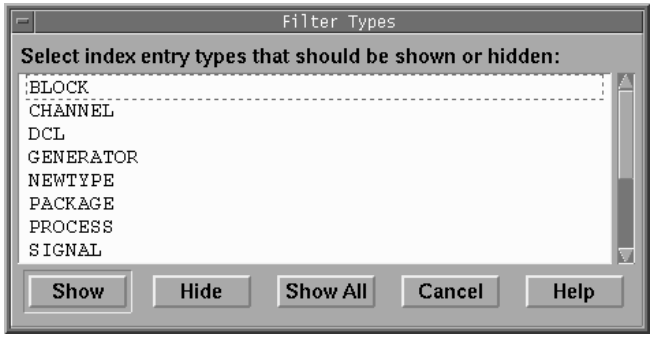


Figure 449: The Filter Types dialog

The figure shows some entity types for an SDL system. See "MSC Icons" on page 2089 for a list of MSC entities.

- *Show*

  Shows entities in the Index Viewer window that have a type that is selected in the dialog. Entities with other types are hidden.

- *Hide*

  Hides entities in the Index Viewer window that have a type that is selected in the dialog. Entities with other types are shown.

- *Show All*

  Shows entities of all types in the Index Viewer window.

An SDT preference, *FilterTypes* can be set to an initial list of entities to show or hide whenever the Index Viewer is opened. Another SDT preference, *ShowSelectedTypes*, determines if the mentioned types should

be shown or hidden. See "FilterTypes" on page 283 in chapter 3, *The Preference Manager*.

Filter Uses

Determines which uses to show/hide. The dialog works in the same way as the dialog for *Filter Types*. The associated SDT preferences are named *FilterUses* and *ShowSelectedUses*.

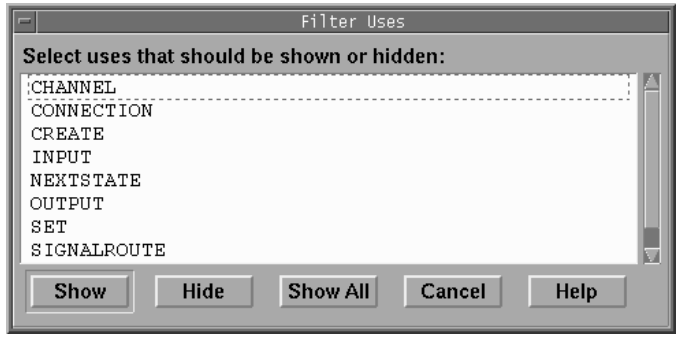


Figure 450: The Filter Uses dialog

Filter Diagrams

Hide/show entities based on which diagram they are defined in. The dialog works in the same way as the dialog for *Filter Types*. The most common use of this operation is to hide definitions from the SDL predefined package. The associated SDT preferences are named *FilterDiagrams* and *ShowSelectedDiagrams*.

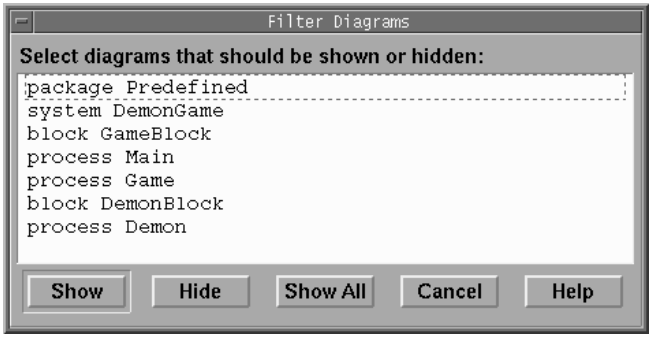


Figure 451: The Filter Diagrams dialog

Set Scale

Issues a dialog where the scale for the Tree window may be set.

Tools Menu

The Tools menu contains the following menu choices:

- *Show Organizer*
  (See "Show Organizer" on page 15 in chapter 1, *User Interface and Basic Operations*.)

- *Search*
- *Search Again*
- *Show Definition*
- *Show Use*

Search

Searches for a visible text in the entity list shown in the Main window.

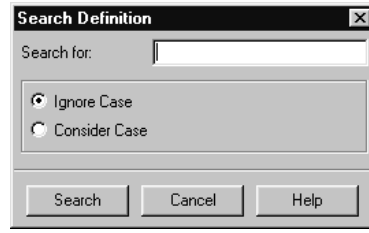


Figure 452: The Search dialog

- *Search for*

  Specifies the text string to search for. As a special case, a GR Reference obtained from for instance an editor may also be specified.

- *Ignore Case, Consider Case*

  These options toggles between a case insensitive and a case sensitive search.

- *Search*

  Starts the search and closes the dialog. The first symbol containing the text is selected. The search starts from the selected symbol, if any, or from the first symbol in the first tree.

If the search reaches the end of the trees, you are asked whether to continue the search from the first tree.

If the text cannot be found among the visible information, you are informed in a confirmation dialog.

Search Again

Searches again for the same text as in the latest search. The behavior is the same as described in "Search" on page 2096.

Show Definition

Opens an editor containing the specified definition of the entity selected in the entity list. The diagram symbol containing the definition becomes selected in the editor.

The name and behavior of this menu choice depends on the number of existing definitions for the selected entity:

- If only one definition exists, that definition is shown in the editor.
- If more than one definition exists, a separate menu choice is available for the first 9 definitions: *Show Definition 1*, *Show Definition 2*, etc. The specified definition is shown in the editor.
- If more than 9 definitions exist, an additional menu choice *Show Definition*... is available. This menu choice opens a dialog in which the number of the definition can be specified.

Show Use

Opens an editor, showing the selected use of an entity. The Show Use operation works in the same way as the *Show Definition* operation.

Popup Menus

The following tables lists the menu choices in the window's popup menus and a reference to the corresponding menu choice in the menu bar.

On the Window Background

Search	"Search" on page 2096.
Search Again	"Search Again" on page 2097.

On an Entity in the Window

Search	"Search" on page 2096.
Search Again	"Search Again" on page 2097.
Show Definition	"Show Definition" on page 2097. The same menu choice variations apply.

Keyboard Accelerators

Apart from the general keyboard accelerators, the following accelerator can be used in the main window:

Accelerator	Reference to corresponding command
`Ctrl+E`	"Show Definition" on page 2097

Quick Buttons

The following quick button is special to the Index Viewer window. The general quick buttons are described in "General Quick-Buttons" on page 24 in chapter 1, *User Interface and Basic Operations*.

Toggle Order
Switches the entity sort order between *name* and *type and name*.

Chapter 46 The SDL Index Viewer

Chapter

47

The SDL Coverage Viewer

The Coverage Viewer shows the coverage of a simulation or validation in the terms of executed transitions or symbols in the system. The information used by the Coverage Viewer can be stored in a coverage file with the file extension `.cov`.

Coverage files can be generated by the SDL Simulator and the SDL Explorer.

This chapter contains a reference manual to the Coverage Viewer; the functionality it provides, its menus, windows and symbols.

Coverage Viewer Windows

The Coverage Viewer has two windows, the Main window and the Coverage Details window. The Main window graphically shows the simulation or validation coverage of transitions or symbols in a tree corresponding to the system structure. The Coverage Details window shows a more detailed coverage chart for a node in the tree.

The Main window is opened when the Coverage Viewer is started, whereas the Coverage Details window is not opened initially. The general characteristics of these windows are described in chapter 1, *User Interface and Basic Operations*.

The Main Window

The Main window shows a graphical tree representing the diagram structure of the simulated or validated system. The tree can either present *Transition Coverage* or *Symbol Coverage*; the mode is controlled by a user option. When the Coverage Viewer first opens a coverage file, the preference *ShowTransitions* is used to determine the default mode.

At the top of the drawing area, above the root of the tree, is a text identifying the type of coverage tree (transition/coverage) and the name of the current coverage file.

In the tree, the different SDL diagram types use the same icons as in the Organizer's main window. For an explanation of all other icons, see "SDL Icons" on page 2086 in chapter 46, *The SDL Index Viewer*.

Transition Coverage

When the tree shows transition coverage, the lowest level in the tree consists of transitions, represented by a small SDL input symbol. Both states and transitions are visible below the process and procedure diagrams in the tree structure. The start transitions are treated as all other transitions, even though they are represented in the tree by an SDL start symbol.

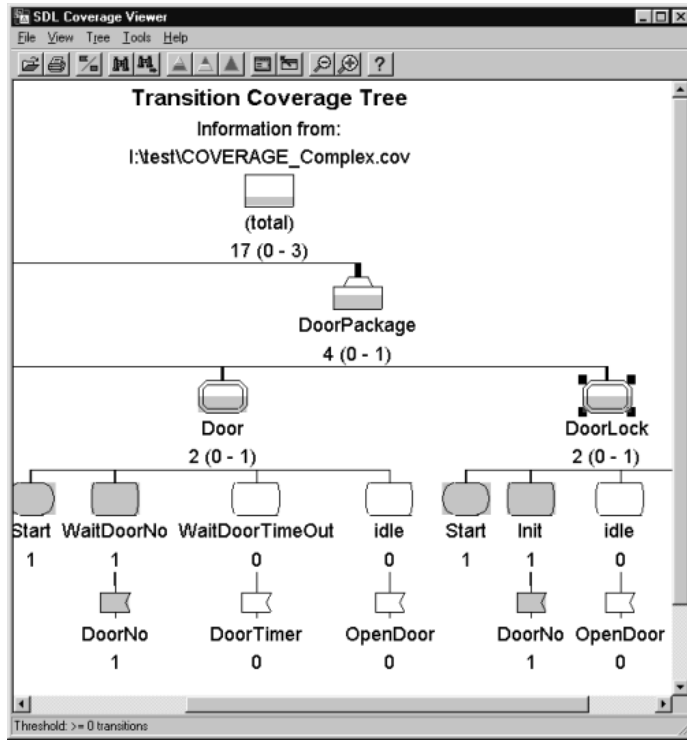


Figure 453: Transition coverage in the Main window

Symbol Coverage

When the tree shows symbol coverage, the lowest level in the tree consists of symbols corresponding to SDL symbols found in flow diagrams.

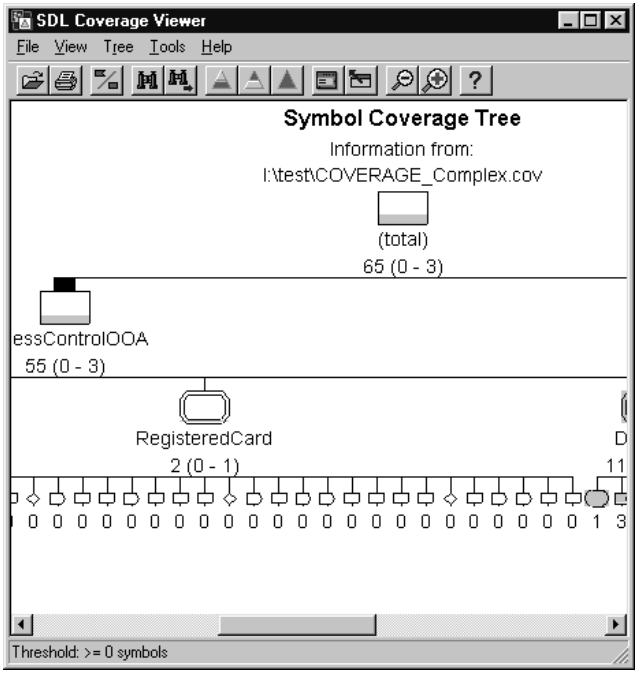


Figure 454: Symbol coverage in the Main window

Nodes

Each node in the tree consists of three parts:

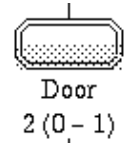


Figure 455: A coverage node

- A graphical symbol indicating the type of node, either:
  - In SDL systems containing packages, an extra root node named “(total)” is introduced, showing the grand total for the system and all packages
  - An SDL diagram (for instance, Figure 455 shows a process type symbol)
  - An SDL state
  - A transition (start transition or input of signal/timer)
  - Any symbol that can be found in SDL flow diagrams

  The graphical notation uses the same symbols as in SDL, where applicable. See also “SDL Icons” on page 2086 in chapter 46, *The SDL Index Viewer*.
- The name of the node is presented as text below the symbol (in Figure 455, the name of the process type is “Door”). An exception to this is the symbols at the lowest level in a symbol coverage tree. These symbols are shown without a name.
- The number of executed transitions/symbols associated with the node is presented below the name. For symbols above the lowest level of the tree, a range is also shown indicating the number of times the least and most executed transition/symbol below the node

has been executed. The process type symbol in Figure 455 reads "2 (0 - 1)" which means that:

- Two transitions/symbols have been executed.
- The least frequently executed transition / symbol in process type Door has been executed zero times.
- The most frequently executed transition / symbol in process type Door has been executed once.

Line Thickness

The number of executed transitions/symbols shown below the node symbol also controls the thickness of the vertical line connecting the node with its parent node.

The line thickness thus gives a visual clue about how much the different branches of the tree have been executed, expressed as a relative number. The thicker the line, the more the tree branch has been executed.

Filling Level

In each node symbol, the number of transitions/symbols that have been executed at least once in relation to the total number of transitions/symbols in that node is also displayed. This is done by filling the node symbol with a gray pattern, showing the percentage level (number of executed divided by total number).

The higher the filling level, the more transitions/symbols have been executed. Figure 455 for instance shows that 50% of the total number of transitions/symbols have been executed.

Dashed Nodes

The visibility condition determines if nodes should be shown or not. A node that should not be shown according to the visibility condition can be visible anyway because a child node is shown. To indicate that a visible node does not match the visibility condition, the node is drawn with a dashed line.

The Visibility Condition

When the Coverage Viewer first opens a coverage file, every node in the coverage tree is shown. You can hide and show nodes in two ways, either via the expand/collapse mechanism (see "View Menu" on page 2110), or via the visibility condition.

By using the visibility condition, you can control which nodes should be visible or not. Only those nodes that meet the visibility condition are shown in the tree, and you cannot use the expand mechanism to see nodes that do not meet the visibility condition.

The visibility condition is a threshold value for the number of times a transition/symbol is executed. It is specified with the *Set Visibility* menu choice in the *Tree* menu (see "Set Visibility" on page 2112). It can also be changed by quick buttons, as well as the *Increase Tree* and *Decrease Tree* menu choices in the *Tree* menu.

A node in a fully expanded tree is visible either if:

- The number of executed transitions/symbols associated with the node meets the visibility condition.
- At least one of the underlying nodes is visible.

The symbol of a node that meets the second condition but not the first one becomes dashed to indicate that the node itself does not meet the visibility condition.

Single and Double Clicks

When you select a node in the main window, the Coverage Details window is updated and displays the coverage chart for the selected node. The Coverage Details window must be opened by the *Show Details* menu choice in the *Tools* menu.

To double-click on a symbol in the main window opens an SDL Editor with a diagram that corresponds to the selected symbol. See "Show in Editor" on page 2115 for more information.

Quick Buttons

The following quick buttons are special to the Main window. The general quick buttons are described in "General Quick-Buttons" on page 24 in chapter 1, *User Interface and Basic Operations*.

Coverage Type
Switches between a transition coverage tree and a symbol coverage tree in the Main window.

Show Bottom
Changes the appearance of the coverage tree to show only those transitions/symbols that have been executed the **least** number of times, i.e. having a number of executions exactly equal to the **lowest** number of available executions. This command changes the visibility condition.

Show Top
Changes the appearance of the coverage tree to show only those transitions/symbols that have been executed the **most** number of times, i.e. having a number of executions exactly equal to the **highest** number of available executions. This command changes the visibility condition.

Show Whole Tree
Changes the appearance of the coverage tree to show all transitions/symbols. The visibility condition is reset to zero or more (>=0).

Show Details Window
Pops up (opens/raises) the Coverage Details window.

The Menu Bar

File Menu

The *File* menu contains the following menu choices:

- *Open*
  (See "Open" on page 9 in chapter 1, *User Interface and Basic Operations*.)
- *Open Directory*
- *Merge*
- *Save*
  (See "Save" on page 11 in chapter 1, *User Interface and Basic Operations*.)
- *Print*
  (See *"Printing from the SDL Suite" on page 315 in chapter 5, Printing Documents and Diagrams*.)
- *Exit*
  (See "Exit" on page 15 in chapter 1, *User Interface and Basic Operations*.)

Open Directory

This menu choice is used to merge information from all coverage files (`*.cov`) in one directory.

> **Note:**
> It is only possible to merge coverage files created from the same SDL system, i.e. the *Open Directory* operation will fail if there are coverage files from different SDL systems in the same directory.

A directory selection dialog is opened. The first coverage file in the specified directory is opened, and all other coverage files in the specified directory are merged. The resulting coverage information is presented in the main coverage window.

Merge

Reads an additional coverage file about the same system and merges the coverage information in it with the current coverage information. A File Selection Dialog is opened with the file filter set to `*.cov`. The contents of the main window is augmented by the additional information read from the selected file.

It is not possible to merge coverage files with another system.

View Menu

The View menu contains the following menu choices:

- *Expand*
- *Expand Substructure*
- *Expand All*
- *Collapse*
- *Window Options*
- *Set Scale*

Expand

Expands the selected node one level down.

Expand Substructure

Expands the selected node as much as possible.

Expand All

Expands all nodes as much as possible.

Collapse

Collapses the selected node, i.e. hides all child objects. This menu choice is named *Collapse All* if no node is selected. Invoking the *Collapse All* menu choice collapses the top node.

Window Options

Sets options for which parts of the Main window to show. The dialog controls whether to show the tool bar and the status bar.

Set Scale

Issues a dialog where the scale for the main window may be set.

Tree Menu

The Tree menu contains the following menu choices:

- *Tree Options*
- *Symbol Options*
- *Line Options*
- *Increase Tree*
- *Decrease Tree*
- *Set Visibility*

Tree Options

Sets options for which type of coverage tree to show in the main window. The dialog controls whether to show either:

– *Transition Coverage*
– *Symbol Coverage*

Symbol Options

Sets options for how to show execution numbers for the nodes in the coverage tree. The dialog controls whether to show the numbers for

– *Every node*,
– the *Bottom nodes* only, or
– *Do not show numbers* at all.

Line Options

Sets options for controlling the line thickness of the nodes in the coverage tree.

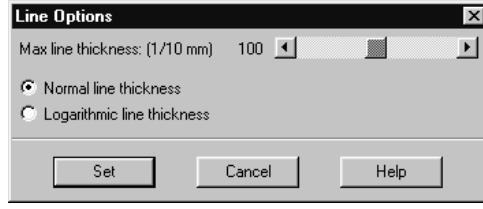


Figure 456: The Line Options dialog

- *Normal line thickness*
- *Logarithmic line thickness*

  The dialog controls whether to show *Normal line thickness* or *Logarithmic line thickness*, i.e. the thickness is proportional either to the executed number of transitions/symbols, or to the logarithm of the same number.

- *Max line thickness*

  With a slide bar, you may also specify the *Max line thickness*, measured in 1/10 mm. The possible range of values is 1–200. The default is 60, which is the thickness of the line connecting the top node of the tree.

Increase Tree

Shows more of the coverage tree by calculating a *visibility threshold* (see "The Visibility Condition" on page 2107) that makes at least one more symbol become visible.

Decrease Tree

Shows less of the coverage tree by calculating a *visibility threshold* (see "The Visibility Condition" on page 2107) that makes at least one more symbol become hidden.

Set Visibility

Sets options to control the visibility condition of the coverage tree (see "The Visibility Condition" on page 2107).

Figure 457: The Set Visibility dialog

The dialog controls:

- *Threshold*

  The value which may be set with the slide bar,

- *Show symbols executed >= threshold*
- *Show symbols executed <= threshold*

  The option whether to make nodes visible that are executed more than or less than the threshold value.

Tools Menu

The Tools menu contains the following menu choices:

- *Show Organizer*
  (See "Show Organizer" on page 15 in chapter 1, *User Interface and Basic Operations*.)

- *Search*
- *Search Again*
- *Show Coverage*
- *Show in Editor*
- *Show Details*

Search

Searches for a visible node name in the coverage tree shown in the Main window.

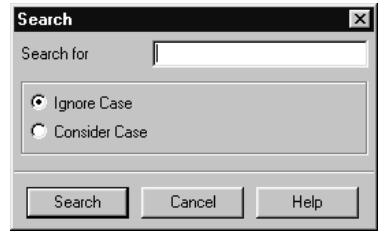


Figure 458: The Search dialog

- *Search for*

  Specifies the text string to search for. As a special case, a GR Reference obtained from for instance an editor may also be specified.

- *Ignore Case, Consider Case*

  These options toggles between a case insensitive and a case sensitive search.

- *Search*

  Starts the search and closes the dialog. The first symbol containing the text is selected. The search starts from the selected symbol, if any, or from the first symbol in the tree.

If the search reaches the end of the tree, you are asked whether to continue the search from the first node.

If the text cannot be found among the visible information, you are informed in a confirmation dialog.

Search Again

Searches again for the same text as in the latest search. The behavior is the same as described in "Search" on page 2113 above.

Show Coverage

Opens an SDL Editor for all the transitions/symbols that meet a certain threshold condition in the tree below the selected symbol. The diagram symbols corresponding to the transitions/symbols become selected in the SDL Editor.

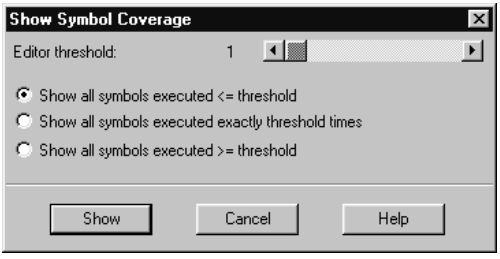


Figure 459: The Show Coverage dialog

The dialog controls the *Editor threshold* value with a slide bar, as well as whether to show transitions/symbols that are executed, either:

- less than
- equal to
- more than

the threshold value.

The *Show* button displays all diagram pages containing transitions or symbols that meet the threshold condition, one page at a time, with a confirmation dialog between each page.

Transitions (represented by the input symbol) or symbols matching the threshold condition are selected and colored with a temporary background color. The temporary background color can be removed with SDL editor > View > Temporary Colors > Remove. The temporary background color is also removed when the diagram is unloaded from the editor.

There is a *No Dialog* button in the confirmation dialog that can be used to avoid displaying the confirmation dialog for each page with symbols to show.

Note:

The editor threshold condition specified in this dialog does not affect the visibility condition. It only controls which transitions/symbols to show in an SDL Editor.

Show in Editor

Opens an SDL Editor with a diagram that corresponds to the selected symbol. If the selected symbol is located in a process or procedure diagram, the corresponding diagram symbol becomes selected in the diagram. Only applicable on transition/symbol icons at the lowest level in the tree, including start states.

Show Details

Opens or raises the Coverage Details window, showing a coverage chart for the selected node.

Popup Menus

The following tables lists the menu choices in the main window popup menus and a reference to the corresponding menu choice in the menu bar, or the corresponding quick button.

On the Main Window Background

Show Bottom	“Show Bottom” on page 2108.
Show Top	“Show Top” on page 2108.
Show Whole Tree	“Show Whole Tree” on page 2108.
Show Details	“Show Details” on page 2115.
Expand All	“Expand All” on page 2110.
Collapse All	Collapses the whole tree, i.e. only the top node becomes visible.

On a Node in the Coverage Tree

Show in Editor	“Show in Editor” on page 2115.
Expand	“Expand” on page 2110.
Expand Substructure	“Expand Substructure” on page 2110.
Collapse	“Collapse” on page 2110.

Keyboard Accelerators

Apart from the general keyboard accelerators, the following accelerator can be used in the main window:

Accelerator	**Reference to corresponding command**
`Ctrl+E`	“Show in Editor” on page 2115

Coverage Details Window

The Coverage Details window presents more detailed coverage information for the node selected in the Main window. It contains a coverage chart for transitions or symbols, depending on what type of tree is shown in the main window. Above the chart is a text identifying:

- The type of coverage chart
- The type and name of the node
- The total number of executed transitions/symbols.
- The number of covered transitions/symbols.
- The number of not covered transitions/symbols.

The Coverage Details window is updated when the selection in the Main window is changed. The Coverage Details window is updated to show the coverage chart for the node selected in the Main window.

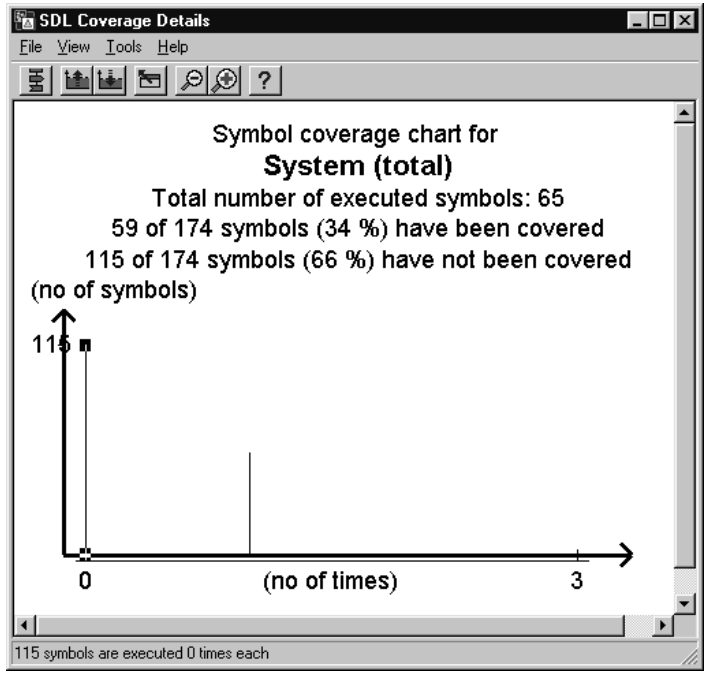


Figure 460: Coverage Details window

The Coverage Chart

The coverage chart shows how many transitions/symbols in the node that have been executed a certain number of times:

- The horizontal axis denotes the number of times a transition/symbol is executed; the range of the axis is the same as the range shown for the node in the Main window.
- The vertical axis denotes the number of transitions/symbols that has been executed a specific number of times; the range of the axis cannot generally be determined from the values shown for the node in the Main window.

For each value along the horizontal axis (no of times), a vertical bar shows how many transitions/symbols have been executed that specific number of times. The vertical axis scale is adjusted to include the highest vertical bar, but this can be changed by using the *Chart Options* menu choice in the *View* menu (see "Chart Options" on page 2120), or the quick buttons.

The left most bar is automatically selected when a new coverage chart is displayed. The selection can be moved by using the left and right arrow keys. Double clicking on a bar in the chart executes the *Show in Editor* command, see "Show in Editor" on page 2120.

Just below the horizontal axis is the *visibility line*. It underlines the bars that correspond to the visible nodes in the coverage tree in the Main window. "Visible" in this case means visible according to the visibility condition. Nodes that are hidden by collapsing a parent node are still regarded as visible in this context.

Quick Buttons

The following quick buttons are special to the Coverage Details window. The general quick buttons are described in "General Quick-Buttons" on page 24 in chapter 1, *User Interface and Basic Operations*.

Show in Editor
Corresponds to the description in "Show in Editor" on page 2120.

Increase Vertical Scale
Increases the vertical scale of the coverage chart by decreasing the value of the upper range of the vertical axis. Those bars that "overflows" the range of the vertical axis are shown with an up arrow at the top of the bars to indicate that the bars in reality are higher.

Decrease Vertical Scale
Decreases the vertical scale of the coverage chart by increasing the value of the upper range of the vertical axis. The reverse of the *Increase Vertical Scale* quick button above.

Show Coverage Tree
Pops up the Coverage Viewer's main window (the Coverage Tree window).

The Menu Bar

File Menu

The *File* menu contains the following menu choices:

- *Print*
  (See *"Printing from the SDL Suite" on page 315 in chapter 5, Printing Documents and Diagrams*.)
- *Close*
  (See "Close" on page 14 in chapter 1, *User Interface and Basic Operations*.)

View Menu

The *View* menu contains the following menu choices:

- *Window Options*
- *Chart Options*
- *Set Scale*

Window Options

Sets options for which parts of the Coverage Details window to show. The dialog controls whether to show the tool bar and the status bar.

Chart Options

Sets options controlling the appearance of the coverage chart.

The dialog controls the value of the upper limit of the vertical axis in the chart and whether to:

- *Show visibility line*
- *Hide Visibility line*

Set Scale

Issues a dialog where the scale for the Coverage Details window may be set.

Tools Menu

The *Tools* menu contains the following menu choices:

- *Show in Editor*
- *Show Coverage Tree*

Show in Editor

Opens an SDL Editor for all the transitions/symbols of the selected vertical bar in the coverage chart. The diagram symbols corresponding to the transitions/symbols become selected in the SDL Editor.

A confirmation dialog is displayed between each diagram page.

Show Coverage Tree

Raises the Coverage Viewer's main window.

Popup Menu

The following table lists the menu choices in the Coverage Details window popup menu and a reference to the corresponding menu choice in the menu bar.

Show Coverage Tree	“Show Coverage Tree” on page 2120.

Keyboard Accelerators

Apart from the general keyboard accelerators, the following accelerators can be used in the Coverage Details window:

Accelerator	**Reference to corresponding command**
`Ctrl+E`	“Show in Editor” on page 2120

Chapter 47 The SDL Coverage Viewer

Chapter

48

Complexity Measurements

Complexity measurements can be generated as a means to analyze the characteristics of a system.

General

It is possible to generate a number of complexity measurements when running the tool. The generated measurements are provided as raw data in a text file with comma as delimiter between fields. Viewing and manipulating this data can be performed using a spreadsheet tool.

The information in the complexity file can be used in many ways, for example:

- A project leader can obtain overall information about how a system evolves, by comparing different complexity files.
- By identifying the "most complex" parts of a system, a project leader can get help about how to allocate both development and test resources.

The Complexity File

The complexity file will have the name `<system name>.csv` and will be generated at the same time and in the same place as the `.xref` file (the target directory).

The `.csv` format is a "standard" spreadsheet format containing comma separated values. Many spreadsheet tools recognize this format, to read the file properly the list separator in the regional setting has to be "," (comma).

In some countries ";" (semicolon) is used as separator. If the separator is a semicolon the file should be imported as a text file, so the separator can be explicitly specified.

Generating the complexity file

The complexity file is generated during the analysis phase according to the following setup:

1. In the Analyzer dialog, select **Semantic Analysis** and click **Details.**
2. In the dialog displayed, select **Generate a complexity measurement file** and enter a file name.

Other ways of generating a complexity file

There are alternate ways of generating a complexity file:

- The environment variable SDT_COMPLEXITY_INFO can be set to generate a complexity file. In that case it is a prerequisite that an .xrf (cross references file) also has to be generated.

  As the calculation of the execution paths (see "Execution Paths" on page 2128) is the most complex seen from the implementation side and at the same time is the calculation that might take some time, this calculation can be turned off by setting the environment variable SDT_NO_COMPLEXITY_PATH_INFO. The figures for this field in the file will then be 0. The recommendation, however, is to leave this calculation turned on, until something seems to go wrong.

- The Analyzer commands *Set-ComplexityMeasurement* and *ComplexityMeasurement-File* can be used for creating a complexity file.

  See "Set-ComplexityMeasurement" on page 2486 in chapter 54, *The SDL Analyzer* and "ComplexityMeasurement-File" on page 2478 in chapter 54, *The SDL Analyzer* for more information.

Complexity File Contents

There is one line in the file for each:

- System, System Type, Package
- Block, Block Type, Block Substructure
- Process, Process Type
- Service, Service Type
- Procedure (not simple procedures, >= 2 states or >= 50 symbols)

Each line contains first the *declaration level*. This is 1 for the system and for packages. Units defined in the system have level 2 and so on.

The lines then include the *entity class* and the *name of the unit*, followed by some complexity numbers. The complexity numbers are given both for the unit itself (local) and for the unit including all its subunits (total).

The lines are sorted so that all subunits are placed **before** the unit where they are defined.

Complexity Numbers

Declarations

This is the number of declarations in the unit. The following entity classes contribute to the declaration count:

- System, System Type
- Block, Block Instance, Block Type, Block Substructure
- Channel, Signal Route, Gate
- Process, Process Instance, Process Type
- Service, Service Instance, Service Type
- Procedure, Operator (user defined), Remote Procedure, Remote Variable
- Sort, Syntype, Generator, Synonym
- Variable, View, Formal Parameter
- Signal, Timer, Signal List

States

This number gives the number of distinct states in the unit. The start symbol is not counted as a state.

This number does not correspond to the number of graphical state symbols, as the same state can be mentioned in several graphical state symbols and as many states can be mentioned in the same graphical state symbols.

Transitions

This number gives the number of transitions defined in the unit. The start transition is considered as a transition. A transition starts with the symbol after a start, input, enabling condition, or continuous signal symbol, and ends with a nextstate, stop, return, or join symbol.

This means for example that the transition starting with a task in the following is counted as one transition:

```
state a,b,c;
  input s1,s2,s3;
    task ....;
    ....;
    nextstate c;
```

Symbols

This is the number of symbols in the transitions in the unit. State, start, input, continuous signal, and enabling conditions are not counted. More precisely the following symbols are counted:

- task
- create
- call
- output
- set
- reset
- decision
- nextstate
- stop
- return
- join

A compound statement in a task or defining an operator or procedure is counted as one symbol.

The count is not directly connected to the graphical symbols, but rather to the syntactical entities. A task is, for example, counted as one symbol independently how it is divided between a task symbol, a text extension, and a comment symbol.

Statements

This is an extension of the count of symbols. Each of the following is also counted individually:

- Assignments in a task
- Signals sent in an output
- Timers handled in a set or reset
- Variables exported in an export
- Statements in a compound statement

Execution Paths

This is the number of possible different paths to go from a transition start (start, input, continuous signal) to a transition end (stop, nextstate, return). This figure is not completely correct but will in most cases give a good estimate.

The following details should be noted:

- Break and Continue statements in compound statements are ignored.
- Loops (joins leading backwards and for statements) are considered to be executed one time.

Maximum Statement Depth

This figure gives the statement depth, that is the nesting of statements. In graphs it is the decision symbol that contributes, while in compound statements it is the compound statement, the if statement, the decision statement, and the for statement that contribute.

Chapter

49 *The SDL Simulator*

This chapter is a reference to the simulator user interface.

For a guide to how to use the simulator, see chapter 50, *Simulating a System*.

The Simulator Monitor

A simulator generated by the SDL Suite consists of two main parts; the application itself (the simulated SDL system), and an interactive monitor system. The monitor system is the interface between the user and the simulated system. For more information on the structure of a simulator and the features of the simulator monitor, see "Structure of a Simulator" on page 2234 in chapter 50, *Simulating a System*.

Monitor User Interfaces

Two different user interfaces are provided for the simulator monitor, a textual and a graphical.

The textual interface only allows commands to be entered from the keyboard, using the syntax described in the section "Syntax of Monitor Commands" on page 2132.

The graphical interface still allows commands to be entered from the keyboard in the same way, but also provides buttons, menus and dialogs for easy access to commands and other features.

The textual interface is invoked by executing the generated simulator directly from the operating system prompt. This is called running a simulator in *textual mode*. When started, the simulator responds with the following text:

```
Welcome to SDL SIMULATOR. Simulating system <system>

Command :
```

Another prompt may appear if the SDL system contains external synonyms. For more information, see "Supplying Values of External Synonyms" on page 2240 in chapter 50, *Simulating a System*.

> **Note:**
>
> Before a simulator can be run in textual mode **on UNIX**, a command file must be executed from the operating system prompt. The file is called `telelogic.sou` or `telelogic.profile` and is located in the binary directory that is included in the user's path.
> For csh-compatible shells: **`source <bin dir>/telelogic.sou`**
> For sh-compatible shells: **`. <bin dir>/telelogic.profile`**

The graphical interface, known as the *Simulator UI*, runs in a separate window. It is started from the Organizer by selecting *SDL > Simulator UI* from the *Tools* menu. The Simulator UI is described in "Graphical User Interface" on page 2199.

If a file called `siminit.com` exists in the current directory an implicit Include-File command will be done on this file at startup.

Activating the Monitor

Commands can be issued to the interactive monitor system when it becomes active. The simulator's monitor system becomes active:

- When the simulator is started.
- When the last command was Next-Transition or Next-Visible-Transition and the transitions initiated by this command have completed.
- When the last command was Next-Symbol or Step-Symbol and one SDL symbol has been executed.
- When the last command was Next-Statement or Step-Statement and one SDL statement has been executed.
- When the last command was Finish and the currently executing procedure has returned.
- When the last command was Proceed-Until and the value of the simulation time is, for the first time, equal to the time given as a parameter to the command.
- When the last command was Proceed-To-Timer and all transitions up to the next timer output have been executed.
- Immediately before a transition matching a transition breakpoint set by the command Breakpoint-Transition.
- Immediately before a symbol matching a symbol breakpoint set by the command Breakpoint-At.
- Immediately after an output symbol that contains an output matching an output breakpoint set by the command Breakpoint-Output.

- Immediately after a symbol or assignment statement where a variable is changed matching a variable breakpoint set by the command Breakpoint-Variable.
- When there is no transition that can be executed, that is, the system is completely inactive. (**On UNIX**, if no environment is present and the command Go-Forever was issued.)
- Immediately after a symbol that included an SDL semantic error.
- In the Simulator UI, when the *Break* button is clicked in the *Execute* button module; in textual mode, when `<Return>` is pressed during the output of trace information. The interactive monitor then becomes active directly after the current symbol has been executed.

Note:

No other characters may be typed before `<Return>` **is pressed**.

Syntax of Monitor Commands

The monitor system waits for commands from the user by issuing the following prompt:

```
Command :
```

The syntax to be used for the commands and their parameters are described in the following.

Command Names

A command name may be abbreviated by giving sufficient characters to distinguish it from other command names. A special character, the hyphen (-), is used to separate command names into distinct parts. Any part may be abbreviated as long as the command name does not become ambiguous.

Consider, as an example, the command name List-Process. The command name may be typed **`List-P`** or **`L-P`**. However, if only **`List`** is typed, the monitor system will respond with the message

```
Command was ambiguous, it might be an abbreviation
of:
```

followed by a list of all commands starting with “List,” since the command cannot be distinguished from, for example, the commands List-Ready-Queue and List-Timer.

There is no distinction made between upper and lower case letters when matching command names.

Parameters

Command parameters are separated by one or several spaces, carriage returns or tabs. A colon is also accepted between a process name and an instance number, when a PId value is required. If the parameter list following a command name is not complete the interactive monitor system will ask for the missing parameters by prompting with the type of the expected parameters.

Parameters may be abbreviated as long as the parameter value does not become ambiguous. There is no distinction made between upper and lower case letters.

> **Note:**
>
> The SDL keywords **Sender**, **Parent**, **Self** and **Offspring** may **not be abbreviated**.

Underscores in SDL Names

Command parameters that are SDL names may also be abbreviated, as long as the abbreviation is unique. The underscore character ‘_’ is used to separate names into distinct parts in the same way as the hyphen character (-) is used with command names. If a name is equal to the first part of another name, as for example Wait and Wait_For_Me, then any abbreviation of Wait is ambiguous. However, if all characters of the shorter name are given (Wait in this example), this is considered as an exact match and the short name will be selected.

The abbreviation facility means that the introduction of underscores in SDL names simplifies simulation. If, for example, you were considering if two signals should be named GenerateOn and GenerateOff or if they should be named Generate_On and Generate_Off, the last alternative is preferable for simulations, as the abbreviations G_On and G_Off are likely to be unique.

Matching of Parameters

When a (possibly abbreviated) parameter name is entered and is to be matched with an unabbreviated name, only names in the entity class of interest are considered. That is, if the monitor expects a process name as parameter, only the names denoting process types will be part of the search for the full name.

Signal names and timer names are in the same entity class; process formal parameters and process variables are in the same entity class; each other type of name (process, procedure, state, block,...) is in an entity class of its own.

Knowledge of previous parameters is used to narrow the search for a given parameter name. Consider for example the command Output-Via, which takes two parameters, a signal and a channel. The channel name is then only searched for among the channels the given signal can be sent via.

Qualifiers

Names can still cause problems, if, for example, there are two process types with the same name in two different blocks, or if the system and a block contain signal definitions with the same name.

In the first situation the process name will always be ambiguous and in the second case the system's signal will always be used. To solve cases like this, qualifiers with the same syntax as in SDL can be used. To reach a process P defined in block B in system S, the following notations can be used:

```
System S / Block B P
<<System S / Block B>> P
```

The words "system," "block," "process," "procedure," and "substructure" in the qualifier **may not be abbreviated**, while all names of, for example, blocks and processes may be abbreviated according to the usual rules. It is only necessary to give those parts of the qualifier that make the qualifier path unique (this is an extension of the qualifier concept in SDL). The slash (/) in a qualifier may be omitted and replaced by one or more spaces. The angle brackets that are part of the qualifier when printed may be omitted when entering the qualifier.

Default Parameters

Some command parameters may be omitted, indicating a default value for the parameter. To accept a default parameter value at the prompt, just press <Return>.

If the parameter is given on the same line as the command name and/or other parameters, type a hyphen '-' to indicate a default parameter value.

Example 324: Default Parameters in Simulator Command

Consider as an example the command Breakpoint-Transition, which takes at least eight parameters. To specify default values for all parameters, except the first and fourth parameter:

```
Breakpoint-Transition P - - State1 - - - -
```

Signal and Timer Parameters

When the parameters of a signal instance or a timer instance are to be entered, the interactive monitor system will ask for the parameters one by one, in the same way as for command parameters. The parameters can also be entered directly after the signal or timer name, possibly enclosed by parenthesis.

Example 325: Signal and Timer Parameters in Simulator Command

```
Signal name : S
Parameter 1 (Integer) : 3
Parameter 2 (Boolean) : true
```

The same specification could also be given as:

```
Signal name : S 3 true
```

or as:

```
Signal name : S (3 true)
```

When entering signal parameters it is not necessary to give all parameter values. By entering '-' at the parameter's position, the parameter is given a "null" value (i.e. the computer's memory for the value is set to 0). By entering ')', the rest of the parameters are given "null" values.

Example 326: Signal Parameters in Simulator Command

```
Signal name : S -, true
```

will give the first parameter a “null” value.

```
Signal name : S (3, -)
```

will give the second parameter a “null” value. Could also be given as:

```
Signal name : S (3)
```

Errors in Commands

If an error in a command name or in one of its parameters is detected, an error message is printed and the execution of the command is interrupted, that is, a command is either executed completely or not at all. **On UNIX**, this is the only way to cancel a command that has been entered, when not using the simulator’s graphical user interface.

Context-Sensitive Help

By typing a question mark (‘?’), a context-sensitive help is obtained. The monitor will respond to a ‘?’ by giving a list of all allowed values at the current position, or by a type name, when it is not suitable to enumerate the values. After the presentation of the list, the input can be continued.

If no default value exists for the requested parameter, the list of all allowed values is also printed if the user simply presses `<Return>` at the prompt. In these cases, there is no need to type ‘?’.

Input and Output of Data Types

The syntax of literals of the predefined data types is in most cases obvious and follows the SDL definition of literals. There are, however, some extensions that are described below. As an option, it is also possible to use the ASN.1 syntax for values. On input the monitor system can manage both value notations, while there are commands in the monitor to select the type of output to be produced (SDL-Value-Notation and ASN1-Value-Notation).

Integer, Natural Values

The format of integers conforms exactly with the SDL and the ASN.1 standard, that is, an integer consists of a sequence of digits, possibly preceded by a '+' or '-'. However, with the command Define-Integer-Output-Mode it is possible to define the base of integers on output (decimal, hexadecimal, octal), which also affects how they may be entered. Hexadecimal values are preceded with "0x", and octal values are preceded with '0' (a zero).

Boolean Values

Boolean values are entered (and printed) as `true` and `false`, using upper or lower case letters. Abbreviation is allowed on input. In ASN.1 mode the value is printed in capital letters (`TRUE`, `FALSE`).

Real Values

The SDL literal syntax of real values has been extended to include the E-notation for exponents, in the same way as in many programming languages.

Example 327: Real Values in SDL Syntax ----------

The real number $1.4527 * 10^{24}$ can be written `1.4527E24`

The real number $4.46 * 10^{-4}$ can be written `4.46E-4`

The syntax for real values in ASN.1 is easiest described by an example:

Example 328: Real Values in ASN.1 Syntax ––––––––––––––––––––

```
{mantissa 23456, base 10, exponent -3}
```

This is the value 23.456.

Time, Duration Values

The format for Time and Duration values follows the SDL standards, i.e. real values without E-notation, with one extension. On input time values can either be absolute or relative to NOW. If the time value is given without a sign an absolute time value is assumed, while if a plus or minus sign precedes the value, a value relative to NOW is assumed.

Example 329: Time Values in SDL Syntax ––––––––––––––––––––

`123.5` is interpreted as 123.5

`+5.5` is interpreted as NOW + 5.5

`-8.0` is interpreted as NOW - 8.0

Character Values

Character values are entered and printed according to the SDL standard, including the literals for the non-printable characters.

Example 330: Character Values in SDL Syntax ––––––––––––––––––

```
'a', '3', '''', ' ', NUL, DEL
```

On input, the quotes may be omitted for all characters except ‘ ’ (space) and ‘[‘. A ‘should be entered as ‘’’.

Charstring Values

Charstring values can be entered and printed according to the SDL standard, that is, a single quote (’) followed by a number of characters followed by a single quote. Any quote (’) within a charstring has to be given twice. On output a non-printable character within a charstring is

printed as a single quote followed by the character literal followed by a single quote.

The ASN.1 syntax for Charstring is similar to the SDL syntax. The delimiter character ‘ (single quote) is however replaced by the character “ (double quote).

Example 331: Charstring Values in SDL Syntax ––––––––––––––––––

`''`	An empty string
`'abc'`	A string of three characters
`'a'NUL'c'`	The second character is NUL

––

PId Values

Apart from the value null, which is a valid PId value, a PId value consists of two parts, the name of the process type and an instance number, which is an integer greater than 0.

The first process instance of a process type that is created will have instance number 1, the second that is created will have instance number 2, and so on. The syntax is Name:No, where Name is the process name and No is the instance number.

On input the process name and the instance number may, as an alternative, be separated by one or more spaces, if the command parameter is a PId value. In the same circumstances the instance number is not necessary (and will not be prompted for) if there is only one process instance of the process type. If, however, the command parameter is a unit that might be a process type or a process instance, only a colon (‘:’) is allowed between the process name and the instance number and the colon must follow directly after the process type name. Examples of such situations are the unit parameter in Set-Trace and Signal-Log.

On output a PId value may be followed by a plus sign (‘+’), which indicates that the process instance is dead; that is, has executed a stop action. The plus sign is chosen as it is reminiscent of the ‘†’ character.

Bit

The Bit sort contains two values, 0 and 1. This syntax is used for input and output.

Bit_String

For Bit_String values the following syntax is used:

```
'0110'B
```

The characters between the two single quotes must be 0 or 1. On input the syntax for Octet_String, see below, can also be used.

Octet

The syntax used for an octet value is two HEX digits. Examples:

```
00   46   F2   a1   CC
```

The characters 0-9, a-f, and A-F are allowed.

Octet_String

The syntax for Octet_String is the following

```
'3A6F'H
```

Each pair of two HEX values in the string is treated as an Octet value. If there is an odd number of characters an extra 0 is inserted last in the string.

Object_Identifier

The sort Object_Identifier is in SDL treated as a String(Natural). This means that the syntax, in case SDL value notation is used, will be:

```
(. 2, 3, 11 .)
```

On input the items in the list should be separated by a comma and/or spaces - tabs. If ASN.1 value notation has been selected, the syntax will be:

```
{ 2 3 11 }
```

On input the items in the list should be separated by a comma and/or spaces - tabs.

Enumerated Values

Types that in SDL are defined as an enumeration of possible values can be entered and printed using the literals of the SDL data type definition. On input, the literals can be abbreviated as long as they are unique.

STRUCT Values

A struct value is entered and printed as the two characters "(." followed by a list of components followed by the two characters ".)". The components should, on input, be separated by a comma and/or a number of spaces (or carriage returns or tabs). Example:

```
(. 23, true, 'a' .)
```

If ASN.1 syntax is used, the component names will also be present. Example:

```
{ Comp1 2, Comp2 TRUE, Comp3 'a' }
```

On input the components using ASN.1 syntax may come in any order. Components not given any value will have the value 0, whatever that means for the data type.

On input the components using ASN.1 syntax may come in any order.

Optional components that are not present, will not be printed. In SDL syntax that means an empty position between two commas.

#UNION Values

A #UNION value uses, in SDL value notation, the same syntax as a struct value. However, only the tag value and the active component is printed. In ASN.1 value notation, #UNION values use the Choice value syntax described below.

Choice Values

The syntax for a choice value is `ComponentName:ComponentValue`. If, for example, a choice contains a component C1 of type integer, then

```
C1:11
```

is a valid choice value.

#UNIONC Values

The syntax for a #UNIONC value is ! `ComponentName ComponentValue` or -> `ComponentName ComponentValue`. If, for example, a #UNIONC contains a component C1 of type integer, then

```
! C1 11
```

is a valid #UNIONC value. However, this syntax will not be used when printing, since the value contains no tag and hence it is not possible to know which component to use.

Array Values

An array value is entered and printed as “(:” followed by a list of components followed by a “:)”. The components should, on input, be separated by a comma and/or a number of spaces (or carriage returns or tabs). Note that there should also be a space between the last component and the terminating “:)”. In ASN.1 syntax, ‘{’ and ‘}’ are used as delimiters.

There are two syntaxes for array components depending on the implementation that the SDL to C Compiler has selected for the array implementation. This selection is described in *“Array” on page 2671 in chapter 56, The Cadvanced/Cbasic SDL to C Compiler*. These easiest way to determine which syntax to use on input is to look at a variable of the array sort.

- If an array is a simple array (i.e. index type is a simple type with one range condition and a limited range), the SDL syntax for an array value is according to the following example:

  ```
  (: 1, 10, 23, 2, 11 :)
  ```

  If ASN.1 value notation is selected, replace “(:” and “:)” by ‘{’ and ‘}’.

- If the array is a general array, the a syntax according to the following example should be used:

  ```
  (: (others:2), (10:3), (11:4) :)
  ```

  This would mean that for index 10 the value is 3, for index 11 the value is 4, and for all other indexes the value is 2. On input the commas, the parenthesis, and the colons in the components may be replaced by one or more spaces (or carriage returns or tabs).

For simple arrays the second syntax is also accepted. If the first syntax is used for simple arrays it is not mandatory to enter all values for the array components; by entering ":)" or "}" the rest of the components are set to a "null" value (i.e. the computer's memory for the value is set to 0).

String Values

A string value starts with "(." and ends with ".)". The components of the string is then enumerated, separated with commas. Example:

```
(. 1, 3, 6, 37 .)
```

On input the commas can be replaced by one or more spaces (or carriage returns or tabs). In ASN.1 syntax, '{' and '}' are used as delimiters instead of "(." and ".)".

Powerset Values

A powerset value starts with a '[' and ends with a ']'. The elements in the powerset is then enumerated, separated with commas. Example:

```
[ 1, 3, 6, 37 ]
```

On input the commas can be replaced by one or more spaces (or carriage returns or tabs).

Bag Values

A bag value starts with a '{' and ends with a '}'. The elements in the bag is then enumerated, separated with commas. Example:

```
{ 1, 3, 6, 37 }
```

On input the commas can be replaced by one or more spaces (or carriage returns or tabs). If the same value occurs more than once, then this value is in SDL syntax not enumerated several times. Instead, the number of occurrences is indicated after the value. Example

```
{ 1, 3:4, 6:2, 37 }
```

This is identical to

```
{ 1, 3, 3, 3, 3, 6, 6, 37 }
```

In ASN.1 syntax, each member is given explicitly according to the last example. On input items are separated by comma and/or one or more

spaces. It is also allowed to mention the same value several times, with or without a number of items each time.

Ref Values

There are two possible syntaxes for Ref values. Either the pointer address as a HEX value, or the value of the data area referenced by the pointer. The value Null is printed as `Null` in both syntaxes. In the monitor system two commands are available to determine the syntax to be used (REF-Address-Notation and REF-Value-Notation). On input both syntaxes are allowed independently of what syntax that has been selected.

Example of the address notation:

```
HEX(23A20020)
```

In the value notation the keyword `new()` is used to indicate a reference to a value. If, for example, the following data types are available in SDL

```
newtype Str struct
  data integer;
  next StrRef;
endnewtype;

newtype StrRef Ref(Str)
endnewtype;
```

then a value of the StrRef type can look like the examples below:

```
Null
new( (. 1, Null .) )
new( (. 1, new( (. 2, Null .) ) .) )
```

The last example is a linked list with two elements. To handle recursive data structures, e.g. graphs, a special syntax is used: `old n`, where `n` is an integer. Example:

```
new( (. 1, new( (. 2, old 1 .) ) .) )
```

The notation `old 1` in the example above means a reference to the same data area as the first `new()` notation refers to. So here we have two structs with next pointers referring to the other struct value. `Old n` would refer to the n:th `new()` seen from the left of the expression.

The same applies for Own and ORef values.

Monitor Commands

This section provides an alphabetical listing of all available commands in the simulator monitor. Simulator UI commands are described in "SimUI Commands" on page 2225.

@ (Keyboard Polling)

Parameters:

(None)

The monitor will repeatedly look at the keyboard for a carriage return, indicating an immediate break in the execution of the current transition. Using the command @, this facility can be turned off. Each time the command @ is entered the monitor toggles between having the keyboard polling for carriage return on and off. At start up keyboard polling is on.

> **Note:**
>
> **No other characters** may be typed together with the carriage return. If some other characters are typed by mistake, please delete them before typing the carriage return.

To turn keyboard polling off is useful in some situations, for example if a user wants to paste a sequence of commands into the monitor. If the sequence contains an empty line this might cause an unwanted interrupt.

? (Interactive Context Sensitive Help)

Parameters:

(None)

The monitor will respond to a '?' (question mark) by giving a list of all allowed values at the current position, or by a type name, when it is not suitable to enumerate the values. After the presentation of the list, the input can be continued.

Typing '?' at the prompt level gives a list of all available commands, after which a command can be entered. For further help, see the command Help.

ASN1-Value-Notation

Parameters:

(None)

The value notation used in all outputs of values is set to ASN.1 value notation. See also the command SDL-Value-Notation.

Assign-Value

Parameters:

```
[ '(' <PId value> ')' ]
<Variable name> <Optional component selection>
<New value>
```

The new value is assigned to the specified variable in the process instance, service instance, or procedure given by the current scope. Sender, Offspring, and Parent may also be changed in this way, but their names may not be abbreviated.

It is, in a similar way as for the command Examine-Variable, possible to handle components in structured variables (struct, strings, array, and Ref) by appending the struct component name or a valid array index before the value to be assigned. Nested structs and arrays can be handled by entering a list of index values and struct component names.

If a PId is given within parenthesis the scope is temporarily changed to this process instance instead.

Breakpoint-At

Parameters:

```
<SDT reference> <Optional breakpoint commands>
```

A breakpoint is defined at the symbol specified by the SDT reference. If the execution reaches the symbol, the monitor becomes active immediately before that symbol. SDT references may be obtained by using the *Show GR Reference* command in an SDL Editor, see "Show GR Reference" on page 2027 in chapter 43, *Using the SDL Editor*.

The <Optional breakpoint commands> parameter can be used to give monitor commands that should be executed when the breakpoint is triggered. Monitor commands should be separated by `" ; "`, i.e. space semicolon space.

Breakpoint-Output

Parameters:

```
<Signal name>
<Receiver process name> <Receiver instance number>
<Sender process name> <Sender instance number>
<Counter>
<Optional breakpoint commands>
```

A breakpoint is activated and a breakpoint condition is defined. If a breakpoint condition is matched by an output, the monitor becomes active immediately after the symbol containing the output. The breakpoint condition defines one or several outputs and is specified by the parameters. Any of the parameters may be omitted, which implies that any value will match the missing fields in the breakpoint condition. Initially no breakpoints are active.

The <Counter> parameter is used to indicate how many times the breakpoint condition should be true before the monitor should become active. Default value for this parameter is 1, which means that the monitor should be activated each time the breakpoint condition is true.

The <Optional breakpoint commands> parameter can be used to give monitor commands that should be executed when the breakpoint is triggered. Monitor commands should be separated by `" ; "`, i.e. space semicolon space.

Breakpoint-Transition

Parameters:

```
<Process name> <Instance number> <Service name>
<State name> <Signal name> <Sender process name>
<Sender instance number> <Counter>
<Optional breakpoint commands>
```

A breakpoint is activated and a breakpoint condition is defined. If a breakpoint condition is matched by a transition, the monitor becomes active immediately before that transition is started. The breakpoint condition matches one or several transitions and is specified by the parameters. Any of the parameters may be omitted, which implies that any value will match the missing fields in the breakpoint condition. Initially no breakpoints are active.

The <Counter> parameter is used to indicate how many times the breakpoint condition should be true before the monitor should become active. Default value for this parameter is 1, which means that the monitor should be activated each time the breakpoint condition is true.

The <Optional breakpoint commands> parameter can be used to give monitor commands that should be executed when the breakpoint is triggered. Monitor commands should be separated by `" ; "`, i.e. space semicolon space.

Breakpoint-Variable

Parameters:

```
<Variable name> <Optional breakpoint commands>
```

A breakpoint is defined on the specified variable in the process instance given by the current scope. If the variable's value is changed, the monitor becomes active immediately after the symbol or assignment statement where the value is changed. The value is only checked between symbols and between assignment statements in tasks.

The breakpoint is also triggered when the variable no longer exists, i.e. the PId containing the variable is stopped or the procedure containing the variable has reached its end. In this case, the breakpoint is automatically removed.

The <Optional breakpoint commands> parameter can be used to give monitor commands that should be executed when the breakpoint is triggered. Monitor commands should be separated by `" ; "`, i.e. space semicolon space.

Call-Env

Parameters:

```
(None)
```

The Call-Env command performs one call of the function `InEnv`. See the command Start-Env for more details.

> **Note:**
>
> This command and the commands Start-Env and Stop-Env are only available when the SDL to C Compiler is used to generate applications.

Call-SDL-Env

Parameters:

```
(None)
```

The command makes one call to the function that looks for incoming signals from the SDL Suite communication mechanism. This means that all signals that have come to the simulator will be sent to their appropriate receiving process instances. See the command Start-SDL-Env for more details.

Cd

Parameters:

```
<Directory>
```

Change the current working directory to the specified directory.

Clear-Coverage-Table

Parameters:

```
(None)
```

This command is used to reset the coverage table to 0 in all positions, which means restart counting coverage from now.

Close-Signal-Log

Parameters:

```
<Entry number>
```

Stops the signal log with the specified entry number and closes the corresponding log file; see the command Signal-Log for more details. Entry numbers assigned by the command List-Signal-Log should be used.

Command-Log-Off

Parameters:

```
(None)
```

The command log facility is turned off; see the command Command-Log-On for details.

Command-Log-On

Parameters:

```
<Optional file name>
```

The command enables logging of all the commands given in the monitor. The first time the command is entered a file name for the log file has to be given as parameter. After that any further Command-Log-On commands, without a file name, will append more information to the previous log file, while a Command-Log-On command with a file name will close the old log file and start using a new file with the specified name.

Initially the command log facility is turned off. It can be turned off explicitly by using the command Command-Log-Off.

The generated log file is directly possible to use as a file in the command Include-File. It will, however, contain exactly the commands given in the session, even those that were not executed due to command errors.

The concluding Command-Log-Off command will also be part of the log file.

Connect-To-Editor

Parameters:

```
(None)
```

This command will create a new *Breakpoints* menu in the SDL Editor and tell the SDL Editor to show all graphical breakpoints. If no SDL Editor is opened, the menu will appear when an SDL Editor is opened the next time. The commands given in the new menu will only be handled in those simulations that are connected. See also the Disconnect-Editor command.

Create

Parameters:

```
<Process name> <Parent's PId value>
<Process parameters>
```

A process instance of the specified process type is created. If the parent PId is not equal to null, Offspring is set in the specified parent instance. If the number of instances of the specified process type is greater than or equal to the maximum number of concurrent instances, the user has to verify the create action.

Define-Continue-Mode

Parameters:

```
"On" | "Off"
```

Defines whether the execution of the simulation shall continue after a command is given. The default is “Off”.

Define-Integer-Output-Mode

Parameters:

```
"dec" | "hex" | "oct"
```

Defines whether integer values are printed in decimal, hexadecimal or octal format. In hexadecimal format the output is preceded with “0x”, in octal format the output is preceded with ‘0’ (a zero).

On input: if the format is set to hexadecimal or octal, the string determines the base as follows: After an optional leading sign a leading zero indicates octal conversion, and a leading “0x” hexadecimal conversion. Otherwise, decimal conversion is used.

The default is “dec”, and no input conversion is performed.

Define-MSC-Trace-Channels

Parameters:

```
"On" | "Off"
```

Defines whether the env instance should be split into one instance for each channel connected to env in the MSC trace. The default is “Off”.

Detailed-Exa-Var

Parameters:

```
(None)
```

When printing structs containing components with default value, these values are explicitly printed after this command is given.

Disconnect-Editor

Parameters:

```
(None)
```

This command will remove the connection to the SDL Editor. If this simulation is the only connected to the SDL Editor, it will also remove the *Breakpoints* menu in the SDL Editor. See also the Connect-To-Editor command.

Display-Array-With-Index

Parameters:

```
"On" | "Off"
```

When Display-Array-With-Index is On and you examine an array, the value of the array element is printed with its index. Index is added before the value of the array element. The default is “Off”.

Down

Parameters:

```
<Optional service name>
```

Moves the scope one step down in the procedure call stack. If the current scope is a process containing services, one of the services should be specified. See also the commands Stack, Set-Scope and Up.

Examine-PId

Parameters:

```
[ '(' <PId value> ')' ]
```

Information about the process instance given by the current scope is printed (see the Set-Scope command for an explanation of scope). This information contains the current values of Parent, Offspring, Sender and a list of all currently active procedure calls made by the process instance (the stack). The list starts with the latest procedure call and ends with the process instance itself. If the process contains services, these services will be listed, each with its own procedure call stack.

If a PId is given within parenthesis information about this process instance is printed instead.

Examine-Signal-Instance

Parameters:

```
<Entry number>
```

The parameters of the signal instance at the position equal to entry number in the input port of the process instance given by the current scope are printed (see the Set-Scope command for an explanation of scope). The entry number is the number associated with the signal instance when the command List-Input-Port is used.

Examine-Timer-Instance

Parameters:

```
<Entry number>
```

The parameters of the specified timer instance are printed. The entry number is the number associated with the timer when the List-Timer command is used.

Examine-Variable

Parameters:

```
[ '(' <PId value> ')' ]
<Optional variable name>
<Optional component selection>
```

The value of the specified variable or formal parameter in the current scope is printed (see the Set-Scope command for an explanation of scope). Variable names may be abbreviated. If no variable name is given, all variable and formal parameter values of the process instance given by the current scope are printed. Sender, Offspring, Self, and Parent may also be examined in this way. Their names, however, may not be abbreviated and they are not included in the list of all variables.

Note:

If a variable is exported, both its current value and its exported value are printed.

It is possible to examine only a component of a struct, string or array variable, by appending the struct component name or a valid array index value as an additional parameter. The component selection can handle structs and arrays within structs and arrays to any depth by giving a list component selection parameters. SDL syntax with '!' and "( )" as well as just spaces, can be used to separate the names and the index values.

It is also possible to print a range of an array by giving "FromIndex: To-Index" after an array name. Note that the space before the ':' is required if FromIndex is a name (enumeration literal), and that no further component selection is possible after a range specification.

To see the possible components that are available in the variable, the variable name must be appended by a space and a '?' on input. A list of components or a type name is then given, after which the input can be continued. After a component name, it is possible to append a '?' again to list possible sub components.

To print the value of the data referenced by a Ref pointer it is possible to use the SDL syntax, i.e. the "*>" notation. If, for example, `Iref` is a Ref(Integer) variable then `Iref*>` is the integer referenced by this pointer. If `Sref` is a Ref of a struct, then `Sref*> ! Comp1` is the Comp1 component in the referenced struct. The sequence `*> !` can in the monitor be replaced by `->` (as for example in C).

If a PId is given within parenthesis the scope is temporarily changed to this process instance instead.

Exit

Parameters:

`(None)`

The simulation is terminated. If the command is abbreviated the monitor asks for confirmation. This is the same command as Quit.

Finish

Parameters:

`(None)`

This command will execute the currently executing procedure up to and including its return, i.e. it will finish this procedure. The execution will also be interrupted at the end of the transition. In a process, Finish will behave just like the command Next-Transition.

Go

Parameters:

(None)

The execution of the simulation is resumed. The execution will continue until one of the conditions listed in "Activating the Monitor" on page 2131 becomes true. To stop the execution of transitions, press <Return> (and only this key). The interactive monitor will then become active when the execution of the current SDL symbol is completed.

This command has to be used with care in the interactive monitor, as it might result in executing the simulation program "forever" (if none of the conditions for activating the monitor becomes true). In a catastrophic situation (e.g. an endless loop within an SDL symbol) it is possible to terminate the program by using the way to stop the execution of a program defined in the operating system (<Ctrl+C> **on UNIX**).

Go-Forever

Parameters:

(None)

The command Go-Forever behaves, in most situations, in the same way as the command Go.

For Go-Forever to behave differently than Go, there has to be an environment to the SDL system that can send signals to the SDL system (communicating simulations or environment functions). When the command Go is issued and the SDL system becomes completely idle (no possible transition and no active timer) the monitor becomes active again and is entered. If Go-Forever was used instead of Go, the monitor is not entered in this case; instead the simulation continues to wait for external stimulus. This feature is valuable when a simulation communicates with other simulations or applications.

Help

Parameters:

<Optional command name>

Issuing the Help command without a parameter will print all the available commands. If a command name is given as parameter, this command will be explained.

Include-File

Parameters:

```
<File name>
```

This command provides the possibility to execute a sequence of monitor commands stored in a text file. The Include-File facility can be useful for including, for example, an initialization sequence or a complete test case. It is allowed to use Include-File in an included sequence of commands; up to five nested levels of include files can be handled.

This command also provides the possibility to perform script calling with variable substitution.

In a script file it is possible to refer to variables using $1, $2, ... , $99. The numbered variables refer to the arguments (actual values) given when calling a script using the command Include-File.

Unix shell script expressions like $0, $#, $? and $* are not available to be given as variable references in script files. Letters are not used as reference variables. For example $a, $1b, $ac are not available to be used as reference variables.

The actual values will be given as parameters in the Include-File command after script-file name. It is possible to pass up to 99 actual values. The numbered variables will be substituted by the actual values during execution of commands in the script files.

Arguments should be separated by spaces. $1 will be substituted by arg1, $2 will be substituted by arg2, $3 will be substituted by arg3, and $4 will be substituted by arg4 and so on.

If there is an argument missing, the corresponding reference variable will not get a value, it is up to the user to provide the correct arguments when using the Include-File command.

List-Breakpoints

Parameters:

```
(None)
```

All active breakpoints are listed. Each breakpoint is assigned an entry number that can be used in the Remove-All-Breakpoints and Show-Breakpoint commands.

List-GR-Trace-Values

Parameters:

```
(None)
```

The values of all currently defined GR traces are listed. The list contains the trace unit type (system, block, process, PId), the unit name, and the GR trace value (both numeric and a textual explanation). See also the commands Set-GR-Trace and Reset-GR-Trace.

List-Input-Port

Parameters:

```
[ '(' <PId value> ')' ]
```

A list of all signal instances in the input port of the process instance given by the current scope is printed (see the Set-Scope command for an explanation of scope). For each signal instance an entry number, the signal type, and the sending process instance is given. A '*' before the entry number indicates that the corresponding signal instance is the signal instance that will be consumed in the next transition performed by the process instance. The entry number can be used in the commands Examine-Signal-Instance, Rearrange-Input-Port and Remove-Signal-Instance.

If a PId is given within parenthesis information about this process instance is printed instead.

List-MSC-Log

Parameters:

```
(None)
```

This command returns the current status of the MSC log (off / interactive / batch). See also the commands Start-Interactive-MSC-Log, Start-Batch-MSC-Log and Stop-MSC-Log.

List-MSC-Trace-Values

Parameters:

```
(None)
```

This command returns the current status for the MSC trace parameters. The list contains the trace unit type (system, block, process, PId), the

unit name, and the MSC trace value (both numeric and a textual explanation). See also the commands Set-MSC-Trace and Reset-MSC-Trace.

List-Process

Parameters:

```
<Optional process name>
```

A list of all process instances associated with the specified process type is produced. If no process name is specified all process instances in the system are listed. The list will contain the same details as described for the List-Ready-Queue command.

List-Ready-Queue

Parameters:

```
(None)
```

A list is produced containing the process instances in the ready queue, i.e., those instances that have received a signal that can cause an immediate transition, but that have not yet had the opportunity to execute this transition to its end. If the process contains services, the currently active service is also listed. If a process/service instance has active procedure calls, the current executing procedure instance is also listed. For each instance in the list the following information is given:

1. An entry number (process instances only).
2. An identification of the process/service/procedure instance.
3. The name of its current state. If the state name is followed by a '*', then the process/service/procedure is currently executing a transition starting from this state.
4. The number of signal instances in the input port of the process instance.
5. The signal name of the signal instance that will cause (or has caused) the transition by the process instance. This signal instance is marked with a '*' before the entry number, if the command List-Input-Port is issued. If the transition has started the signal is no longer in the input port.

The process instance, the state, and the signal instance determine uniquely the transition that will be executed by the process instance when it gets permission to do so.

The entry number can be used in the command Rearrange-Ready-Queue.

List-Signal-Log

Parameters:

```
(None)
```

Print information about the currently active signal log; see the command Signal-Log for more details. Each log is assigned an entry number which can be used in subsequent Close-Signal-Log commands.

List-Timer

Parameters:

```
(None)
```

A list of all currently active timers is produced. For each timer, its corresponding process instance and associated time is given. An entry number will also be part of the list, which can be used in the command Examine-Timer-Instance.

List-Trace-Values

Parameters:

```
(None)
```

The values of all currently defined traces are listed. The list contains the trace unit type (system, block, process, PId), the unit name and the trace value (both numeric and a textual explanation). See also the commands Set-Trace and Reset-Trace.

Log-Off

Parameters:

```
(None)
```

The command Log-Off turns off the interaction log facility, which is described in the command Log-On.

Log-On

Parameters:

```
<Optional file name>
```

The command Log-On takes an optional file name as a parameter and enables logging of all the interaction between a simulation program and the user that is visible on the screen. The first time the command is entered, a file name for the log file has to be given as parameter. After that any further Log-On commands, without a file name, will append more

information to the previous log file, while a Log-On with a file name will close the old log file and start using a new file with the specified file name.

Initially the interaction log facility is turned off. It can be turned off explicitly by using the command Log-Off.

News

Parameters:

```
(None)
```

The News command summarizes the changes in the textual monitor user interface from previous releases.

Next-Statement

Parameters:

```
<Optional number of statements>
```

This command is used to step statement for statement through SDL transitions. A statement is the same as a symbol, except that a task symbol may contain several assignment statements; compare with the Next-Symbol command.

Next-Statement steps over procedure calls, i.e. all actions in the procedure will be executed and the monitor will be entered again when the statement after the procedure call is reached; compare with the Step-Statement command.

After making the step(s) the monitor is entered, making it possible to, for example, examine the temporary status of the actual process instance.

> **Note:**
>
> The right hand side of an assignment may contain a value returning procedure call.

Next-Symbol

Parameters:

```
<Optional number of symbols>
```

This command is used to step symbol for symbol through SDL transitions. A symbol may contain several statements; compare with the Next-Statement command.

Next-Symbol steps over procedure calls, i.e. all actions in the procedure will be executed and the monitor will be entered again when the symbol after the procedure call is reached; compare with the Step-Symbol command.

Using the optional integer parameter, a specified number of symbols can be stepped through. Next-Symbol will, however, never step from within one transition into another transition.

After making the step(s) the monitor is entered, making it possible to, for example, examine the temporary status of the actual process instance.

Note:

Join is not considered a symbol by this command.

To determine how far a transition has been executed, the commands Show-Next-Symbol and Show-Previous-Symbol, together with trace printouts and GR traces can be used. The last trace printout is from the last executed symbol (if it has caused a printout, which depends on the trace level), while the symbol next to be executed is selected by the GR trace, if GR trace is on.

Next-Transition

Parameters:

```
(None)
```

The next transition is executed. If real time is used and the next transition is a timer output scheduled in the future (more than a second from now), the command Next-Transition will wait one second, after which the monitor is entered again.

Executing a Next-Transition command within a transition will execute the remaining part of the transition.

Next-Visible-Transition

Parameters:

```
(None)
```

A number of transitions are executed up to and including the next transition with a trace value > 0. For a timer output transition, it is the trace value for the corresponding process instance that is considered.

This command should be used with care in the interactive monitor, as it might result in executing the simulation program “forever” (if no transition with trace>0 is ever executed). To stop the execution of transitions, press `<Return>`. The interactive monitor will then become active when the execution of the current SDL symbol is completed.

Nextstate

Parameters:

```
<State name>
```

The state of the process/service/procedure instance given by the current scope is changed (see the Set-Scope command for an explanation of scope). This command implies that all actions in a nextstate are performed, i.e. recalculation of enabling conditions and continuous signals, and searching for a signal to be received according to the input and save sets. As a consequence the ready queue for process instances may change.

Now

Parameters:

```
(None)
```

The current value of the simulation time is printed.

Output-From-Env

Parameters:

```
-
```

The command will list all signals possible to send into the system, together with receiver. If the signal contains parameters, an empty parenthesis pair will be added after the signal name. Only signals that will trigger a transition directly will be listed. So signals causing an immediate null transition or signals that will be saved are not listed.

By just giving Output-From-Env in the Simulator UI, the possible choices will be given in a dialogue and by selecting one of the signals it is sent into the system. If the signal contains parameters they will be given "null" values.

Output-Internal

Parameters:

```
<Signal name> <Signal parameters>
<Receiver's PId value> <Sender's PId value>
```

The command is used to simulate an output of an internal signal instance between two process instances in the system. The parameter <Sender's PId value> may also denote "env:1". No check is made to ensure that a path of signal routes and channels exists. The monitor responds with either of the three messages below.

```
Signal <name> is not in valid input signal set of
process <name>
```

The signal was not in the valid input signal set of the receiver and can thus not be handled by the receiver. The signal was immediately discarded.

```
Signal <name> was sent to <receiver> from <sender>
```

The signal instance was successfully sent and was placed in the input port of the receiver.

```
Signal <name> caused an immediate null transition
```

The receiving process instance is currently waiting in a state where the specified signal type neither may cause an input operation, nor is saved. The signal instance is immediately discarded (in SDL this is called a null transition, i.e a transition with no actions, leading back to the same state).

Output-None

Parameters:

```
<PId value> <Optional service name>
```

The command is used to send a signal "none" to the specified process instance. If the process instance contains services, one of the services should to be specified as well. The none signal is used to indicate that a spontaneous transition should be initiated in the current state.

Output-To

Parameters:

```
<Signal name> <Signal parameters>
<Receiver's PId value>
```

The command is used to send a signal from the environment of the system to a process instance in the system. It will be checked that a path of channels and signal routes exists from the environment to the specified process instance.

The monitor will respond with either of the messages described below, or by an error message if no path existed.

```
Signal <name> was sent to <receiver> from <sender>
```

The signal instance was successfully sent and is placed into the input port of the receiver.

```
Signal <name> caused an immediate null transition.
```

The receiving process instance is currently waiting in a state where the specified signal type neither may cause an input operation, nor is saved. The signal instance is immediately discarded (in SDL this is called a null transition, a transition with no actions, leading back to the same state).

Output-Via

Parameters:

```
<Signal name> <Signal parameters>
<Optional channel name>
```

The command is used to send a signal from the environment of the system to a process instance in the system via the specified channel. If no channel is given, it is assumed that any channel from the environment to the system may be selected. It will be checked that there exists one process instance that can receive the signal, according to the rules of outputs in SDL.

The monitor will respond with either of the messages below, or by an error message if the number of possible receivers is not equal to one.

```
Signal <name> was sent to <receiver> from <sender>
```

The signal instance was successfully sent and is placed into the input port of the receiver.

```
Signal <name> caused an immediate null transition.
```

The receiving process instance is currently waiting in a state where the specified signal type neither may cause an input operation, nor is saved. The signal instance is immediately discarded (in SDL this is called a null transition, a transition with no actions, leading back to the same state).

Print-Coverage-Table

Parameters:

```
<File name>
```

This command can be used to obtain test coverage information and profiling information. Each time the command is issued a coverage file, with the name specified as parameter, is produced in the work directory

containing the relevant information. The coverage file always reflects the situation from the start of the simulation. To study the information in the coverage file, it can be opened in the Coverage Viewer; see chapter 47, *The SDL Coverage Viewer*.

> **Note:**
>
> The specified file is always overwritten, i.e. there is no confirmation message if an existing file is specified.

The generated file consists of two sections, first a summary with profiling information, containing the number of transitions and the number of symbols executed by each process type. Secondly the file contains detailed information about how many times each symbol and each state - input combination is executed.

Example 332: Profiling Information in Coverage File ––––––––––––––

```
*************** PROFILING INFORMATION **************

2  System DemonGame : Transitions = 13, Symbols = 40
3  Block GameBlock
4  Process Main : Transitions = 3 (23%),
                  Symbols = 10 (25%), MaxQ = 2
4  Process Game : Transitions = 6 (46%),
                  Symbols = 15 (37%), MaxQ = 2
3  Block DemonBlock
4  Process Demon : Transitions = 4 (30%),
                  Symbols = 15 (37%), MaxQ = 1
```

––

This information in Example 332 should be interpreted in the following way:

- As *transitions*, the number of executed input symbols + continuous signal symbols + start symbols are counted.
- As *symbols*, the number of executed process symbols (task, output, decision, set, reset, call, export, stop, return, create, nextstate, input, continuous signal, start) are counted.
- The number of transitions and the number of executed symbols are, as shown in the example above, presented both for the system (total for all processes) and for each process type. The relative numbers for the processes are, of course, relative to the numbers for the system.

- *MaxQ* is the maximum input port queue length for any instance of the process type. Note that start up signals, used to implement create, and continuous signals are counted as signals in MaxQ.
- The numbers at the beginning of each line are the scope level of the that unit. This can be used to determine if, for example, a procedure is defined within or after another procedure.

Note:

To be true, profiling information **execution time** ought to be measured instead of the number of transitions and number of executed symbols, but this information is still very valuable for getting a feeling for the load distribution.

Example 333: Coverage Table in Coverage File ——————————

```
************** COVERAGE TABLE DETAILS **************

2 System DemonGame

3 Block GameBlock

4 Process Main
  1 : Start  #SDTREF(SDL,main.spr(1),131(30,10),1)
  1 : Game_Off  Newgame  #SDTREF(SDL,main.spr(1),137
  1 : Game_On  Endgame  #SDTREF(SDL,main.spr(1),119(
  --------------------
  1 START : #SDTREF(SDL,main.spr(1),131(30,10),1)
  1 INPUT : #SDTREF(SDL,main.spr(1),137(31,44),1)
  1 INPUT : #SDTREF(SDL,main.spr(1),119(56,44),1)
  1 NEXTSTATE : #SDTREF(SDL,main.spr(1),134(34,29),1)
  1 CREATE : #SDTREF(SDL,main.spr(1),140(37,59),1)
  1 TASK : #SDTREF(SDL,main.spr(1),143(34,72),1)
  1 NEXTSTATE : #SDTREF(SDL,main.spr(1),146(35,89),1)
  1 OUTPUT : #SDTREF(SDL,main.spr(1),122(56,59),1)
  1 TASK : #SDTREF(SDL,main.spr(1),125(60,72),1)
  1 NEXTSTATE : #SDTREF(SDL,main.spr(1),128(59,89),1)
```

————————————————————————————————————

For each process type two types of information are given, separated by a line of hyphens:

- The first part contains information about the start transition and each relevant state-input combination: the number of times executed and a GR or PR reference to the associated process symbol. Continuous signals are counted as a separate input.

- The second part contains information about each process symbol: the number of times it has been executed, the type of symbol, and a GR or PR Reference to the symbol.

To identify the process symbols, the command *Show GR Reference* command in the SDL Editor's *Tools* menu can be used for the printed GR references. (See "Show GR Reference" on page 2027 in chapter 43, *Using the SDL Editor*.)

Proceed-To-Timer

Parameters:

```
(None)
```

This command will execute all transitions up to but not including the next timer output. The timer output will not be executed even if it is the next transition.

Proceed-Until

Parameters:

```
<Time value>
```

The execution of the simulation is resumed. The monitor will become active when the value of the simulation time first becomes equal to the time value given as parameter.

Note that relative time values can be given using the '+' sign. Entering "+5.0" as parameter is interpreted as the time value NOW+5.0. See also "Input and Output of Data Types" on page 2137.

Quit

Parameters:

```
(None)
```

The simulation is terminated. If the command is abbreviated the monitor asks for confirmation. This is the same command as Exit.

Rearrange-Input-Port

Parameters:

```
<Entry number> <New entry number>
```

This command is used to change the order of signal instances in the input port of the process instance given by the current scope (see the Set-Scope command for an explanation of scope).

The entry number parameters refer to the numbers assigned by the command List-Input-Port. The signal instance with entry number equal to the first parameter will be moved to a position where it in a subsequent List-Input-Port command will be assigned the entry number given as the second parameter.

Rearrange-Ready-Queue

Parameters:

```
<Entry number> <New entry number>
```

This command is used to change the order in which transitions by process instances are executed. The entry number parameters refer to the entry numbers assigned by the command List-Ready-Queue. The process instance with entry number equal to the first parameter will be moved to a position where it in a subsequent List-Ready-Queue command will be assigned the entry number given as the second parameter.

The Rearrange-Ready-Queue command should be used with care when process priorities are used. The ready queue is then sorted in priority order (see also "Scheduling" on page 2647 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*), which means that a rearrangement might disturb this order. Such a disturbance does not harm, except when a new process should be inserted into the ready queue. In this situation the insert point might not be the proper one. The following insert algorithm is used: Search from the end of the ready queue until a process with higher or equal priority (lower or equal priority value) is found, then insert the new process after the found process. If no process with higher or equal priority is found, then insert the new process first in the ready queue.

REF-Address-Notation

Parameters:

```
(None)
```

REF values (pointers introduced using the generator Ref) are printed as addresses, using the HEX value for the address. The Null value is printed as `Null`. On input, both this syntax and the Value-Notation (see command REF-Value-Notation) can be used.

The same applies for Own and ORef values.

REF-Value-Notation

Parameters:

(None)

REF values (pointers introduced using the generator Ref) are printed as NEW(<the value the pointer refers to>). This is the default syntax for REF values. It means that complete lists or graphs will be printed. Example:

```
NEW( (. 1, NEW( (. 2, Null .) ) .) )
```

To avoid problems in cyclic graphs a special syntax is used if a pointer refers to an address already presented in the output. OLD n, where n is a digit, means a reference to the n:th NEW in the printed value. Example:

```
NEW( (. 1, NEW( (. 2, OLD 1 .) ) .) )
```

The Null value is printed as Null. On input both this syntax and the Address-Notation (see command REF-Address-Notation) can be used.

The same applies for Own and ORef values.

Remove-All-Breakpoints

Parameters:

(None)

This command removes all breakpoints.

Remove-At

Parameters:

<SDT reference>

This command removes all breakpoints at the symbol specified by the SDT reference.

Remove-Breakpoint

Parameters:

<Entry number>

This command removes the breakpoint with the specified entry number. The entry number given by List-Breakpoints should be used.

Remove-Signal-Instance

Parameters:

<Entry number>

The signal instance with the given entry number in the input port of the process instance given by the current scope is removed (see the Set-Scope command for an explanation of scope). The entry number given by List-Input-Port should be used. If this signal was selected for the next transition, the process instance will execute an implicit next-state action.

> **Note:**
>
> Entry numbers are just positions in the input port. The removal of a signal changes the entry numbers of the remaining signals.

Reset-GR-Trace

Parameters:

```
<Optional unit name>
```

The GR trace value of the given unit is reset to undefined. If no unit is specified the GR trace value of the system is reset to undefined. As there always has to be a GR trace value defined for the system, Reset-GR-Trace on the system is considered to be equal to setting the GR trace value to 0. For more information about optional unit names, see the command Set-Trace.

Reset-MSC-Trace

Parameters:

```
<Optional unit name>
```

The MSC trace value of the given unit is reset to undefined. If no unit is specified the MSC trace value of the system is reset to undefined. As there always has to be a MSC trace value defined for the system, Reset-MSC-Trace on the system is considered to be equal to setting the trace value to 0. For more information about optional unit names, see the command Set-MSC-Trace.

Reset-Timer

Parameters:

```
<Timer name> <Timer parameters>
```

The result of the command is exactly the same as if the process instance given by the current scope had executed a reset action. If the reset action causes a timer signal to be removed and this signal was selected for the next transition, the process instance will execute an implicit nextstate action.

Reset-Trace

Parameters:

```
<Optional unit name>
```

The trace value of the given unit is reset to undefined. If no unit is specified the trace value of the system is reset to undefined. As there always has to be a trace value defined for the system, Reset-Trace on the system is considered to be equal to setting the trace value to 0. For more information about optional unit names, see the command Set-Trace.

Restore-State

Parameters:

```
<File name>
```

This command will restore the state of the current simulation to a state given in a file created with the Save-State command. The file name is given as parameter.

The write and read functions for the sorts are used to save and restore the variable values. So if a user has added a sort via the ADT directive, these write and read functions must be consistent.

It is recommended to restart the simulator before giving the Restore-State command. The command is not allowed when the execution is within a transition.

If the SDL system contains the Any construct, the sequence of random numbers will not be the same after the restore.

Only the sdl execution state is restored, no monitor settings are restored. E.g. Show-Previous-Symbol, Show-Next-Symbol does not work directly after a restore. The MSC trace is not restored. Coverage is not restored either.

The ADT package pidlist.pr is not supported, since the PId synonyms that are initialized are not valid after the restore.

Extern C and C++ variables will not be saved and restored since the simulator has no knowledge of them.

Charstar, Voidstar and Voidstarstar variables are not handled.

Since all variables are saved, dangling Ref and ORef variables will cause problems.

Scope

Parameters:

```
(None)
```

This command prints the current scope. See the command Set-Scope for a description of scope.

SDL-Value-Notation

Parameters:

```
(None)
```

The value notation used in all outputs of values is set to SDL value notation. This is the default value notation. See also the command ASN1-Value-Notation.

Save-Breakpoints

Parameters:

```
<File name>
```

This command will save all breakpoints in a file, with the name specified as parameter (in the work directory). It is written as a text file with commands, so the breakpoints can be restored with the Include-File command.

Save-State

Parameters:

```
<File name>
```

This command will save the state of the current simulation in a file, with the name specified as parameter (in the work directory). It is written as a text file with special commands, so the state can be restored with the Restore-State command.

Set-GR-Trace

Parameters:

```
<Optional unit name> <Trace value>
```

The GR trace value is assigned to the specified unit (system, block, process, or process instance). If no unit is specified the GR trace value is assigned to the system. The initial GR trace value of the system is 0, i.e. no GR trace, while it is undefined for all other units. For more information about optional unit names, see the command Set-Trace.

For a description of the possible GR trace values, see "GR Traces" on page 2186.

Set-MSC-Trace

Parameters:

```
<Optional unit name> <Trace value>
```

This command enables the trace of SDL events that take place during the simulation and which can be transformed into events in a Message Sequence Chart. Typically, the events that can be transformed are sending and consumption of signals, and creation and termination of processes.

> **Note:**
>
> Setting the MSC trace value with this command **does not** start the actual logging of MSC events. To do this, the command Start-Interactive-MSC-Log or Start-Batch-MSC-Log is used.

The scope of the trace can be delimited by specifying an optional unit name and a trace value. The general considerations for specifying the unit name for the command Set-Trace are also applicable for Set-MSC-Trace.

Optional Unit Name

- The trace value is assigned to the specified unit (system, block, process, or process instance).
- When specifying a unit name, the scope of trace affects **all process instances** contained in the **underlying SDL structure**.
- Initially, the scope of trace is set on the entire SDL system. So, by default, all events will generate Message Sequence Chart traces. If the unit name is omitted, the trace value is assigned to the system.

Trace Value

- Specifying a trace value of 0 means no trace for the currently specified unit name.
- Specifying a trace value of 1 means trace only if both involved units have a trace value greater than 0.
- Specifying a trace value of 2 means always trace, independent of the trace value of the other involved unit.
- Specifying a trace value of 3 means trace on the block level.

The initial MSC trace value of the system is 1. For more information on MSC trace values, see "Message Sequence Chart Traces" on page 2187.

> **Note:**
>
> Once the logging of Message Sequence Chart traces has been started (using any of the commands Start-Interactive-MSC-Log or Start-Batch-MSC-Log) **modifying the scope of trace may cause unpredictable results**, such as Message Sequence Charts with an unexpected appearance.

Set-Scope

Parameters:

```
<PId value> <Optional service name>
```

This command sets the scope to the specified process, at the bottom procedure call. If the process contains services, one of the services can be given as parameter to the command. A scope is a reference to a process instance, a reference to a service instance if the process contains services, and possibly a reference to a procedure instance called from this process/service. The scope is used for a number of other commands for examining and changing the local properties of a process instance. The scope is automatically set by the execution commands, when entering the monitor, to the next process in turn to execute.

The command Scope prints out the current scope; i.e. the name of the process instance and possibly the service instance and the called procedure instance. See also the commands Stack, Define-MSC-Trace-Channels and Up.

Set-Timer

Parameters:

```
<Timer name> <Timer parameters> <Time value>
```

The result of the command is exactly the same as if the process instance given by the current scope had executed a set action. If the set action causes a timer signal to be removed and this signal was selected for the next transition, the process instance will execute an implicit nextstate action.

Set-Trace

Parameters:

```
<Optional unit name> <Trace value>
```

The trace value is assigned to the specified unit (system, block, process, or process instance). If no unit is specified the trace value is assigned to the system. The initial trace value for the system is 4, while it is undefined for all other units. For a description of the possible trace values, see "Trace Limit Table" on page 2184.

There might, in some cases, be problems in identifying a specific unit. If more than one unit in the system match the, possibly abbreviated, unit name, the first unit found when searched from the system level will be assigned. To make sure the correct unit is assigned, the unit's diagram type, e.g. "process" (unabbreviated), can be introduced before the unit name. If there still are problems, for instance due to the fact that there are several processes with the same name, a qualifier should be inserted immediately before the unit name. An example can be found in "Specifying Unit Names" on page 2254 in chapter 50, *Simulating a System*.

To specify a PId value, a colon and an instance number must follow directly after the process name; see also "Input and Output of Data Types" on page 2137.

Show-Breakpoint

Parameters:

```
<Entry number>
```

This command is only applicable for symbol breakpoints defined with the command Breakpoint-At. The breakpoint with the specified entry number is listed, and the symbol with the breakpoint will be selected in an SDL Editor showing the source GR document. The entry number given by List-Breakpoints should be used.

Show-C-Line-Number

Parameters:

```
(None)
```

This command prints the `.c` file name and line number where the execution of the current SDL transition is suspended. The Text Editor is opened with the cursor positioned on that line in the C source file. The given line number will reference a statement with the following structure:

```
XBETWEEN_SYMBOLS(...)
```

The case label just following this statement is the place where the execution will be resumed when an execute command is given in the monitor.

The Show-C-Line-Number command is mainly thought to be used when the execution is interrupted within an SDL transition. If the command is issued between two transitions, the `XBETWEEN_SYMBOLS` statement immediately before the last Nextstate or Stop operation will be referenced in the printout, together with a warning that this is not the current position.

> **Note:**
>
> Changes made in the monitor, as for example Rearrange-Ready-Queue, will not affect the printout by this command.

Show-Coverage-Viewer

Parameters:

`(None)`

This command starts the Coverage Viewer tool with the current test coverage loaded.

Show-Next-Symbol

Parameters:

`(None)`

The symbol in turn to be executed will be selected in an SDL Editor showing the source GR document. If the simulation is not connected to the SDL Suite or the SDL source document is in SDL/PR, a GR symbol reference or a PR reference will instead be displayed on the screen. The details about GR symbol references are presented in "Dynamic Errors" on page 2190. A PR reference is a file name and a line number. See also "Syntax" on page 917 in chapter 18, *SDT References*.

Note that between the execution of two transitions, i.e. after a Nextstate or Stop operation, no next symbol can be shown. Note also that changes made in the monitor, as for example changing the ready queue using the command Rearrange-Ready-Queue, will not affect the symbol selected by Show-Next-Symbol.

This command uses the same mechanisms to select symbols in the SDL Editor as the GR trace facility, so the same general characteristics as presented in "GR Traces" on page 2186 are also valid for this command.

Show-Previous-Symbol

Parameters:

```
(None)
```

The last executed symbol will be selected in an SDL Editor displaying the source GR document. If the simulation is not connected to the SDL Suite or the SDL source document is in SDL/PR, then a GR symbol reference or a PR reference will instead be displayed on the screen. See the command Show-Next-Symbol for more information.

Show-Versions

Parameters:

```
(None)
```

The versions of the SDL to C Compiler and the runtime kernel that generated the currently executing program are presented.

Signal-Log

Parameters:

```
<Unit name> <File name>
```

Starts logging of signals to a specified file. See also the commands Close-Signal-Log and List-Signal-Log.

The unit name parameter should be either a channel, a signal route, a system, a block, a process type, or a process instance.

- If the unit is a channel or a signal route then all signals sent through the channel or signal route will be logged on the specified file.
- If the unit is a process type or a process instance then all signals sent to or from any instance of the process type or to or from the specified process instance will be logged.
- If the unit is a block or a system then all signals that have relevance for the block or system will be logged, i.e. signals sent within the block or system or signals sent to or from the block or system.

Note that the process “env” is a legal unit in the Signal-Log command. By specifying “env” as unit parameter, the signal interface between the system and the environment is logged. For more information about the unit name parameter, see the command Set-Trace.

Stack

Parameters:

`(None)`

The procedure call stack for the PId/service defined by the scope is printed. For each entry in the stack, the type of instance (procedure/process/service), the instance name and the current state is printed. See also the commands Set-Scope, Define-MSC-Trace-Channels and Up.

Start-Batch-MSC-Log

Parameters:

`<Symbol level> <File name>`

This command starts the logging of SDL events which can be translated into the corresponding MSC events in a log file (see "Mapping Between SDL and MSC" on page 2187). See also the commands Start-Interactive-MSC-Log and Stop-MSC-Log.

The results will be stored in a log file, whose contents are specific for this purpose. That log file can later on be read by a Message Sequence Chart Editor, where its contents are interpreted as a Message Sequence Chart and displayed as such.

The symbol level parameter determines if states and actions should be included in the MSC log. For a description of the possible symbol level values, see "Level of Symbol Logging" on page 2188.

The file name parameter to this command can be any valid file name, although it is recommended to use a file name with the suffix `.mpr`, since this is the default suffix used when reading a log file into a Message Sequence Chart Editor.

Start-Env

Parameters:

`(None)`

When the SDL to C Compiler is used to generate applications, interface functions towards the environment of the SDL system must be provided. The `InEnv` function, which is used to enter signals into the SDL system, is frequently called from the main loop in the process scheduler. During debugging, the polling of the `InEnv` function can turned on and turned off.

The Start-Env command turns on the polling of `InEnv`. At start up of the program polling is turned off. See also the commands Stop-Env and Call-Env.

> **Note:**
>
> This command and the commands Stop-Env and Call-Env are only available when the SDL to C Compiler is used to generate applications.

Start-Interactive-MSC-Log

Parameters:

```
<Symbol level>
```

This command starts the logging of SDL events which can be translated into the corresponding MSC events in a Message Sequence Chart Editor (see "Mapping Between SDL and MSC" on page 2187). See also the commands Start-Batch-MSC-Log and Stop-MSC-Log.

The symbol level parameter determines if states and actions should be included in the MSC log. For a description of the possible symbol level values, see "Level of Symbol Logging" on page 2188.

When the command is issued, the following takes place:

1. An instance of the Message Sequence Chart Editor is started. Each currently existing (SDL) process instance is displayed as an instance, with its instance head and its instance axis.
2. Following the execution of the simulation, each SDL event which is possible to map to an MSC event and which is within the scope of MSC trace which is currently defined will automatically be displayed in the MSC Editor. Each event will cause the insertion point in the MSC Editor to be moved downwards with one step, which provides a feeling of absolute order between events.

Some drawing conventions and default layouting algorithms are used when drawing the automatically generated Message Sequence Chart. These conventions are described in *"Drawing Conventions" on page 1727 in chapter 39, Using Diagram Editors*.

Start-SDL-Env

Parameters:

```
(None)
```

A simulator can communicate with other simulations using SDL signals and the SDL Suite communication mechanism. Signals sent to the environment in one simulation can enter as signals from the environment in another simulator.

This command is used to tell a simulator to start sending signals that are designated to the environment via the SDL Suite communication mechanism and to start looking for incoming signals (polling) from the SDL Suite communication mechanism. See also the commands Stop-SDL-Env and Call-SDL-Env.

This facility is at program start up turned off, and should only be turned on when a simulator should be able to communicate with other simulators (or applications).

> **Note:**
> The command must be given in **both communicating simulators**.

Start-ITEX-Com

Parameters:

`(None)`

This command starts the communication with the TTCN Suite.

Stop-ITEX-Com

Parameters:

`(None)`

This command stops the communication with the TTCN Suite.

Start-SimUI

Parameters:

`(None)`

This command starts a Simulator UI and connects the running simulator to it. After this it is only possible to give commands in the UI.

Start-UI

Parameters:

`(None)`

This command attempts to start the program `sdtenv` in the start directory. The started program is assumed to connect itself to the SDL Suite communication mechanism and can then communicate with the simulation program.

Note:

The command Start-SDL-Env should be given in the simulation program to make it communicate.

Step-Statement

Parameters:

```
<Optional number of statements>
```

This command is used to step statement for statement through SDL transitions. A statement is the same as a symbol, except that a task symbol may contain several assignment statements; compare with the command Step-Symbol.

Step-Statement will step into procedure calls; compare with the Next-Statement command.

After making the step(s) the monitor is entered, making it possible to, for example, examine the temporary status of the actual process instance.

Note:

The right hand side of an assignment may contain a value returning procedure call.

Step-Symbol

Parameters:

```
<Optional number of symbols>
```

This command is used to step symbol for symbol through SDL transitions. A symbol may contain several statements; compare with the Step-Statement command.

Step-Symbol will step into procedure calls; compare with the Next-Symbol command.

Using the optional parameter, a specified number of symbols can be stepped through. Step-Symbol will, however, never step from within one transition into another transition.

After making the step(s) the monitor is entered, making it possible to, for example, examine the temporary status of the actual process instance.

Note:

Join is not considered a symbol by this command.

Stop

Parameters:

```
<PId value> <Optional service name>
```

The specified process instance is stopped. If the process contains services, then either the process can be stopped by giving no service parameter, or one of the services can be stopped. The result of the command is exactly the same as if the process instance or service instance had executed a stop action.

Stop-Env

Parameters:

```
(None)
```

This command turns off the polling of the InEnv function; see the Start-Env command for more details.

Note:

This command and the commands Start-Env and Call-Env are only available when the SDL to C Compiler is used to generate applications.

Stop-MSC-Log

Parameters:

```
(None)
```

This command stops the logging of Message Sequence Chart events (in interactive mode as well as in batch mode). In the case of a batch mode logging, the log file will be closed. See the commands Start-Interactive-MSC-Log and Start-Batch-MSC-Log for more details.

Following this command, it is possible to log the rest of the session on a new file.

Stop-SDL-Env

Parameters:

```
(None)
```

The command turns off the communication mechanism described for the Start-SDL-Env command.

Up

Parameters:

(None)

Moves the scope one step up in the procedure call stack. Up from a service leads to the process containing the service. See also the commands Set-Scope, Stack and Define-MSC-Trace-Channels.

Traces

When a process instance executes actions within a transition, trace information describing the current action might be printed on the screen. The amount of information printed can be selected using the trace commands in the monitor system. A typical trace from a transition containing a few actions is given below.

```
*** TRANSITION START
*      PId    : Demon:1
*      State  : Generate
*      Input  : T
*      Sender : Demon:1
*      Now    : 1.0000
*   OUTPUT of Bump to Game:1
*   SET on timer T at 2.0000
*** NEXTSTATE  Generate
```

The transitions can also be traced in the GR source diagrams. This is discussed in detail under "GR Traces" on page 2186.

Transition Trace

A trace value, which is a non-negative integer, can be assigned to process instances, to process types, blocks, and to the system. The commands associated with traces are Set-Trace, Reset-Trace and List-Trace-Values.

When a process instance starts a transition, the trace value that governs the amount of trace to be printed is computed according to the following algorithm:

1. If a trace value is defined for the process instance executing the transition, that value is used.
2. If not, and a trace value is defined for the process type, that value is used.
3. Otherwise, if a trace value is defined for the block enclosing the process, that value is used.
4. If still no trace value is found, the block structure is followed outwards until a unit is reached which has a trace value defined. The system will always have a trace value.

The hereby computed value is compared with the trace limit for each action executed during the transition. The trace information is printed only

if the trace value is greater or equal to the trace limit for the action. Trace information at trace level 1 is treated specially, see below.

In the table below the trace limits for all the information that can be part of a trace is presented.

Note:

The **trace messages** are **produced** when **actions** defined in the SDL diagrams are **executed**. These messages are not used when monitor commands like Set-Timer, Nextstate, or Stop are entered.

Trace Limit Table

Action	Trace limit
Transition start	2
Parameters of signal in input	6
Output to environment, signal name and receiver	1 (see below)
Output, signal name and receiver	3
Output caused immediate null transition	5
Parameters of signal in output	6
Task	4
Decision, value of expression in decision	4
Procedure start, procedure return	3
Parameters to procedures	6
Create, successful or unsuccessful	3
Parameters to create	6
Set, timer name, time	3
Time less than Now in set, changed to Now	5
Parameters in set	6
Set caused an implicit reset action	5

Action	Trace limit
Reset, timer name	3
Reset caused timer or signal to be removed	5
Parameters to reset	6
Output of timer signal, timer name, receiver, time	2
Output of timer signal caused a null transition	5
Parameters to timer signal	6
Nextstate, state name	3
Null transitions at a nextstate	5
Parameters to signals in null transitions at nextstate	6
Stop	3
Signals discarded at stop	5
Parameters to signals discarded at stop	6
Timers discarded at stop	5
Parameters to timers discarded at stop	6
Export	4

The trace limit table can be summarized as follows:

Trace Limit Table Summary	
0	No trace
1	Trace of signals to environment (environment as seen from the specified unit)
2	Trace of transition start and timer outputs
3	As 2 + trace of important SDL actions
4	As 3 + trace of other SDL actions as well
5	As 4 + result of actions (example: discarded signals)
6	As 5 + Parameters of signals, timers, create actions

Trace information at trace level 1, i.e. trace of signals sent to the environment of the specified unit, is treated in a special way. This information is not printed if the trace value is greater than 1, instead the normal trace of outputs is used.

GR Traces

GR trace is a way to follow the execution of transitions in the GR source diagrams by selecting SDL symbols. It should normally only be used for a small number of processes to limit the amount of information displayed. The GR trace value determines to what degree the execution will be traced, i.e. how often SDL symbols will be selected in the GR diagrams (see below). After a nextstate or stop operation, i.e. between two transitions, the nextstate or stop symbol is still selected.

The commands associated with GR traces are Set-GR-Trace, Reset-GR-Trace and List-GR-Trace-Values.

The GR tracing will take place in a single SDL Editor window, which will show the appropriate GR diagram as the execution progresses. If no SDL Editor is opened, a new editor is started.

Three trace values are possible:

- 0 - No GR trace.
- 1 - When the monitor is entered next, show the next symbol to be executed. No symbols are selected during execution when the monitor is inactive. Thus, the SDL Editor window may not show the diagram containing the symbol currently executed.
- 2 - Follow the execution and show each symbol as it is executed. If the execution is continued into another diagram, this diagram is loaded into the SDL Editor window.

Message Sequence Chart Traces

The commands associated with MSC traces are Set-MSC-Trace, Reset-MSC-Trace and List-MSC-Trace-Values.

Mapping Between SDL and MSC

When executing an SDL system, some of the SDL events can be transformed into a corresponding symbol in a Message Sequence Chart. The mapping rules which govern how SDL events are transformed into MSC symbols, lines and textual elements are described in "Mapping Between SDL and MSC" on page 1726 in chapter 39, *Using Diagram Editors*.

The ITU definition of the MSC language introduces the *instance* concept. An instance is mapped to any instance of an SDL process.

Scope of Trace for Generation of Message Sequence Charts

The scope of MSC trace is the process instances that have a calculated MSC trace value > 0.

Assume that we have set the scope of MSC trace to a part of the SDL system, for instance a block and the underlying structure in terms of processes. When an event takes place in the SDL system, one of three possible situations is possible. Let us assume for the sake of simplicity that the event is the sending of a signal from one process to another process. The three cases and their behavior are as follows:

- First case: The two units (the sending process and the receiving process) are both within the scope of MSC trace. It is easy to transform this into the sending and consuming of a message, and the result will be a trace in a Message Sequence Chart.
- Second case: None of the units is in the scope of MSC trace. This will not result in any trace in a Message Sequence Chart.
- Third case: One and (one only) of the units is in the scope of MSC trace. This case is slightly more complicated. Sending and receiving messages to and from units may be of interest outside the scope of trace.

  The concept *conditional trace* is introduced: A conditional trace will indicate that a message has been sent or received, and that the

receiver or sender is beyond the scope of trace. See also "The Void Instance" on page 1730 in chapter 39, *Using Diagram Editors*.

The MSC trace levels 1 and 2 are used to specify if conditional trace should be presented or not; see the table below. MSC trace level 3 specifies a block level trace.

MSC Trace for Sender	MSC Trace for Receiver	Result
0	0	No trace
	1	No trace
	2	Conditional trace
1	0	No trace
	1	Trace
	2	Trace
2	0	Conditional trace
	1	Trace
	2	Trace

Level of Symbol Logging

The *symbol level* determines the amount of information that should be part of the MSC log. Three symbol levels are possible:

- 0 - Events for signals and timers plus create and stop.
- 1 - As level 0, plus condition symbols for each nextstate.
- 2 - As level 1, plus action symbols for task, decision, call, and return.

Signal Parameter Length

When using MSC trace it is possible to set a maximum parameter trace length by using the environment variable `SDTMSCMAXPARAMLENGTH`.

This can improve performance on simulations using large signal parameters that otherwise could cause the MSC editor to work slowly.

If a parameter exceeds the maximum length it will be truncated to the maximum length with an ending "..." to indicate that the parameter has been truncated.

After the ending "..." there can be possible ending parenthesis, apostrophes or quotes to match initial ones in order to make the signal parameter syntactically correct.

Initial Trace Values

At the start of a simulation, the trace values, the GR trace values and the MSC trace values for all units except the system are undefined. The system trace is 4, the system GR trace is 0, and the system MSC trace is 1.

Dynamic Errors

Violations of the dynamic rules of SDL will cause dynamic errors during the execution of a simulation program. A dynamic error is presented to the user on the screen as an error message. See the example below.

Example 334: Dynamic Error Printout

```
Warning in SDL Output of signal Bump
Signal sent to NULL, signal discarded
Sender: Demon:1
TRANSITION
  Process         : Demon:1
  State           : Generate
  Input           : T
  Symbol          :
#SDTREF(SDL,/usr/tom/demon.spr(1),122(30,55),1)
TRACE BACK
  Process         : Demon
  Block           : DemonBlock
  System          : Demongame
```

The symbol reference given in the `TRANSITION` message should be interpreted as follows:

Item	Text in Example 334	Interpretation
1	`SDL`	Reference to SDL/GR object
2	`/usr/tom/demon.spr`	Reference to the file
3	`(1)`	Page name
4	`122`	The object identity (an unique number assigned by the SDL Editor)
5	`30`	The x-coordinate of the object in mm. The origin of coordinates is the upper left corner of the page.
6	`55`	The y-coordinate of the object in mm
7	`1`	Line number within symbol

By entering the monitor command Show-Previous-Symbol, the symbol that caused the error is displayed.

For more information on references, see chapter 18, *SDT References*.

Dynamic Errors Found by a Simulation Program

```
Error in SDL array index in sort <sort>:
<value> is out of range.
```

Violation of the index range given in an array.

```
Error in assignment in sort <sort>:
<value> is out of range.
```

Violation of the range conditions given in a syntype.

```
Error in SDL Decision: Value is <value>
Entering decision error state
```

The value of the expression in the decision did not match any of the possibilities (answers).

```
Error in SDL Import. Attempt to import from NULL

Error in SDL Import. Attempt to import from stopped
process instance

Error in SDL Import. Attempt to import from the
environment

Error in SDL Import. No process exports this
variable

Error in SDL Import. The specified process does not
export this variable
```

Error in an import statement. Supplementary information about remote variable and exporting processes is also given.

```
Warning in SDL Output of signal <signal>.
No path to receiver, signal discarded
```

An attempt was made to output a signal to a PId expression. There exists, however, no path of channels and signal routes between the sender and the receiver that can convey the signal.

```
Warning in SDL Output of signal <signal>.
No possible receiver found, signal discarded
```

An attempt was made to output a signal without specifying a to PId expression. When all paths or all paths mentioned in a via clause had been examined no possible receiver was found.

```
Warning in SDL Output of signal <signal>.
Signal sent to NULL, signal discarded
```

An attempt was made to output a signal to a PId expression that was null.

```
Warning in SDL Output of signal <signal>.
Signal sent to stopped process instance
```

An attempt was made to output a signal to a PId expression that referred to a process instance which has performed a stop action.

```
Error in SDL View. Attempt to view from NULL

Error in SDL View. Attempt to view from stopped
process instance

Error in SDL View. Attempt to view from the
environment

Error in SDL View: The specified process does not
reveal this variable

Error in SDL View: No process reveals this variable
```

Error in a view statement. Supplementary information about viewed variable and viewing process is also given.

```
Error in SDL Create: Process <process>
More static instances then maximum number of
instances.
```

Obvious!

```
Illegal #UNION tag value for assignment to component
Name.
Tag value is xxx.
```

Attempt to assign a value to a non-active UNION component.

```
Illegal #UNION tag value for access to component
Name.
Tag value is xxx.
```

Attempt to access a non-active UNION component.

```
Error when accessing component Name. Component is
not Present.
```

Attempt to access a non-active optional struct component.

```
Component Name is not active.Present is xxx.
```

Attempt to access a non-active choice component.

```
Dereferencing of NULL pointer.
Pointer assigned new data area at address: HEX(xxx)
```

Attempt to de-reference a Null pointer defined using the Ref generator.

```
User specified error: SDL error expression found
```

Error introduced by the user, by inserting the error expression defined in SDL.

Errors Found in Operators

The errors that can be found in operators defined in the predefined data types are listed below.

```
Error in SDL Operator: Extract! in sort Charstring,
Index out of bounds

Error in SDL Operator: Modify! in sort Charstring,
Character NUL not allowed.

Error in SDL Operator: MkString in sort Charstring,
Character NUL not allowed.

Error in SDL Operator: First in sort Charstring,
Charstring length is 0

Error in SDL Operator: Fix in sort Integer, Integer
overflow

Error in SDL Operator: Last in sort Charstring,
Charstring length is 0

Error in SDL Operator: Substring in sort Charstring,
Charstring length is 0

Error in SDL Operator: Substring in sort Charstring,
Length of substring < 0

Error in SDL Operator: Substring in sort Charstring,
Start index is <= 0

Error in SDL Operator: Substring in sort Charstring,
Start index + length of substring > length of
charstring

Error in SDL Operator: / in sort Integer, Attempt to
divide by 0.

Error in SDL Operator: / in sort Real, Attempt to
divide by 0.0.
```

```
Error in SDL Operator: Rem in sort Integer,
Second operand is 0

Error in SDL Operator: Mod in sort Integer,
Second operand is 0

Error in SDL Operator: Modify! in sort Bit_String,
Index out of bounds.

Error in SDL Operator: Extract! in sort Bit_String,
Index out of bounds.

Error in SDL Operator: First in sort Bit_String,
Bit_String length is zero.

Error in SDL Operator: Last in sort Bit_String,
Bit_String length is zero.

Error in SDL Operator: SubString in sort Bit_String,
Bit_String length is zero.

Error in SDL Operator: SubString in sort Bit_String,
Start is less than zero.

Error in SDL Operator: SubString in sort Bit_String,
SubLength is less than or equal to zero.

Error in SDL Operator: SubString in sort Bit_String,
Start + Substring length is greater than string
length.

Error in SDL Operator: BitStr in sort Bit_String,
Illegal character in Charstring (not 0 or 1).

Error in SDL Operator: HexStr in sort Bit_String,
Illegal character in Charstring (not digit or a-f).

Error in SDL Operator: Modify! in sort Octet,
Index out of bounds.

Error in SDL Operator: Extract! in sort Octet,
Index out of bounds.

Error in SDL Operator: Division in sort Octet,
Octet division with 0.

Error in SDL Operator: Mod operator in sort Octet,
Right operand is 0.

Error in SDL Operator: Rem operator in sort Octet,
Right operand is 0.

Error in SDL Operator: BitStr in sort Octet,
An Octet should consist of not more than 8
characters 0 or 1.
```

```
Error in SDL Operator: HexStr in sort Octet,
An Octet should consist of 2 HEX values.

Error in SDL Operator: HexStr in sort Octet,
Illegal character in Charstring (not digit or a-f).

Error in SDL Operator: Modify! in sort Octet_String,
Index out of bounds.

Error in SDL Operator: Extract! in sort
Octet_String,
Index out of bounds.

Error in SDL Operator: BitStr in sort Octet_String,
An Octet_String should consist of 0 and 1.

Error in SDL Operator: HexStr in sort Octet_String,
Illegal character in Charstring (not digit or a-f).
```

Apart from these error message each instantiation of generator String will introduce similar error messages as for the Charstring sort. Note that `Object_Identifier` is a predefined sort which is an instantiation of String.

Action on Dynamic Errors

After a dynamic error, the execution of the simulation program is continued until the current symbol is ended. The interactive monitor will then become active. The following actions will be taken:

- If the error was an output error the signal will not be sent.
- If the error was a decision error the process instance will immediately be placed in an *decision error state*. It can only be removed from this state by using the Nextstate command in the monitor. The input port will not be affected when the decision error state is entered. All signals sent to a process instance in a decision error state will be saved in the input port.
- If the error occurred during the creation of static process instances, i.e. the initial number of instances is greater that the maximum number of instances, an error message is given and the number of instances specified by initial number of instances are created.
- If the error occurred during an import or view action a data area of the correct size containing zero in all positions is returned.

- If the error was found in a range condition check during an assignment, the variable to the left of the assignment operator will be assigned the computed value, although it is out of bounds.
- If the error was found during a range check of an array index the index value will be changed to be the lowest value of the index type. This means that the corresponding C array will never be indexed out of its bounds.
- If the error occurred during selection of an optional component in a struct or when selecting a component in a #UNION or Choice, an error message is given and the operation is perform anyhow.
- If a Null pointer (Ref) is de-referenced, an new data area of correct size is allocated containing zeros. This data area is assigned to the pointer. After the error message the statement containing the de-referencing is performed.
- If the error occurred within an expression the operator that found the error returns a default value and the evaluation of the expression is continued. The default values returned depend on the result type of the operator and are given in section *"Default Values" on page 2674 in chapter 56, The Cadvanced/Cbasic SDL to C Compiler*.

Assertions

A user can define his own run-time errors or assertions. Basically the run-time library only provides an appropriate C function that can be called to print out an error message. This function is assumed to be used in #CODE directives in TASKs according to the following example:

Example 335: Assertion in C Code ----------

```
TASK  '' /*#CODE
#ifdef XASSERT
  if (#(I) < #(K))
    xAssertError("I is less than K");
#endif
*/ ;
```

The `xAssertError` function, which has the following prototype:

```
extern void xAssertError ( char *Descr )
```

takes a string describing the assertion as parameter and will produce an SDL run-time error similar to the normal run-time errors. The function is only available if the compilation switch XASSERT is defined. For the standard libraries this is true for all libraries except the *Application Library*.

Run-time Prompting

For some special SDL constructs, an SDL simulator is not able to continue executing without prompting the user for necessary input. The execution continues when the user has entered an allowed value.

Decision with Any

If an SDL DECISION is encountered with an ANY question, the user will be prompted which path to execute:

```
Decision with ANY
1 (go path)
1 ? (show path)
2 (go path)
2 ? (show path)
3 (go path)
3 ? (show path)
Enter path :
```

At this point, the user must enter a path value (integer) within the allowed range. If GR trace is enabled, the user may enter a path value followed by a '?' to see the corresponding path in an SDL Editor.

Informal Decision

If an SDL DECISION is encountered with informal text, the user will be prompted which path to execute:

```
Informal decision: 'Question1'
'answer1'  1
'answer2'  2
'ELSE'  3
Enter path :
```

At this point, the user must enter a path value (integer) within the allowed range. If GR trace is enabled, the user may enter a path value followed by a '?' to see the corresponding path in an SDL Editor.

Unimplemented Operator

If an expression is encountered involving an operator that is not implemented, the user will be prompted for the value to be returned depending on the parameters:

```
Operator op in sort s is called
Parameter 1: 50
Enter value(s) :
```

At this point, the user must enter a value in the resulting sort to be the returned value of the operator when called with the listed parameter values.

Graphical User Interface

This section describes the appearance and functionality of the graphical user interface to the simulator monitor (SimUI). Some user interface descriptions general to all tools can be found in chapter 1, *User Interface and Basic Operations*. These general descriptions are not repeated in this chapter.

Starting the SimUI

A new SimUI is started by selecting *SDL > Simulator UI* from the *Tools* menu in the Organizer. When the SimUI is started, a number of definition files are read, controlling the contents of the main window and some status windows. See "Definition Files" on page 2221 for more information.

No simulator is started automatically by the SimUI in this way. The user must start a simulator by selecting *Open* from the *File* menu, as stated in the text area of the main window, or by using the *Open* quick button.

A simple way to generate a simulator, start the SimUI and open the simulator is to click the *Simulate* quick button in the Organizer.

When a simulator is started, a file selection dialog may be opened if the SDL system contains external synonyms. For more information, see "Supplying Values of External Synonyms" on page 2240 in chapter 50, *Simulating a System*.

The Main Window

The main window provides a text area (which displays output from the monitor system), an input line (used for entering and displaying textual command line input to the monitor system and the SimUI) and a button area with button modules (with buttons for execution of monitor and SimUI commands).

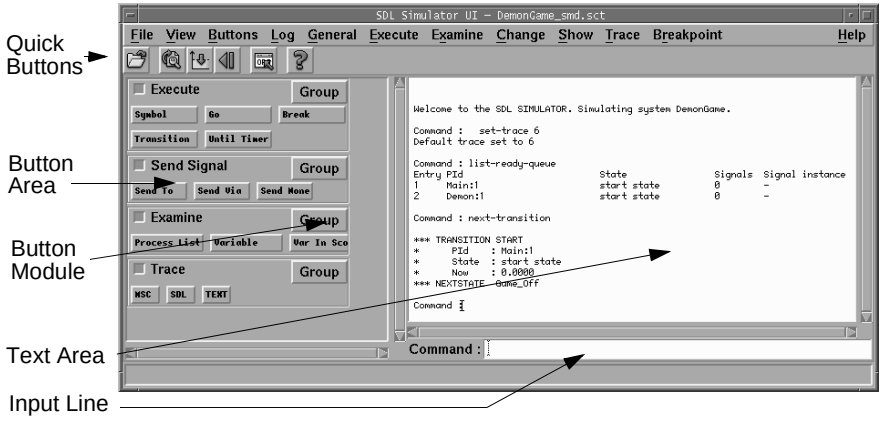


Figure 461: The main window

The Text Area

The *text area* displays all text output from the monitor system, including user prompts, error messages and command results.

Commands cannot be entered in this area, but a command given on the input line or through the use of the command buttons is echoed after the displayed prompt:

```
Command :
```

The Input Line

The *input line* is used for entering and editing commands from the keyboard. For information on available monitor commands, see "Monitor Commands" on page 2145. There are also special SimUI commands that are not sent on to the simulator monitor, see "SimUI Commands" on page 2225.

The 100 latest commands entered on the input line are saved in a history list. The history list can be traversed by using the `<Up>` and `<Down>` keys on the input line.

When `<Return>` is pressed anywhere on the input line, the complete string is saved in the history list and is moved to the text area. The command is then executed.

The input line also has an associated popup menu with the choices *Undo*, *Command*, *Save*, *Go to Source* and *Add to watchwindow*:

- Both *Undo* and *Command* open a dialog with all commands entered so far. In the *Undo* case, all the commands to redo can be selected, and the default is to redo all but the last one, which is equivalent to an Undo of the last command. In the *Command* case, only one of the commands to re-execute can be selected. For more information, see "Undo/Redo Commands" on page 2247 in chapter 50, *Simulating a System*.
- *Save* will save all monitor commands issued by the user so far to a command file. It opens a file selection dialog, in which the name of a command file is selected (**on UNIX** preferably with the suffix `.com`).
- *Go to Source* will go to the SDT reference selected in the text area.
- *Add to watchwindow* will add the variable selected in the text area to the watch window.

Parameter Dialogs

If a command entered on the input line requires additional user input (i.e. parameter values), the information will automatically be asked for in a dialog:

- Parameter values that are file names are selected in File Selection Dialogs.
- Parameter values of enumeration type are presented in lists, from which the value can be selected (see Figure 462).
- Other parameter values are prompted for in simple text input dialogs. In these dialogs, the button *Default value* will enter a "null" value for the parameter in the input field.

Each parameter dialog has an *OK* button for confirming the value and a *Cancel* button for cancelling the command altogether. Some parameters have a default value that does not have to be specified. In this case, an empty value or '-' is accepted for the default value.

Enumeration Type Parameter Dialogs

Additional functionality is available in the dialog for enumeration type parameters:

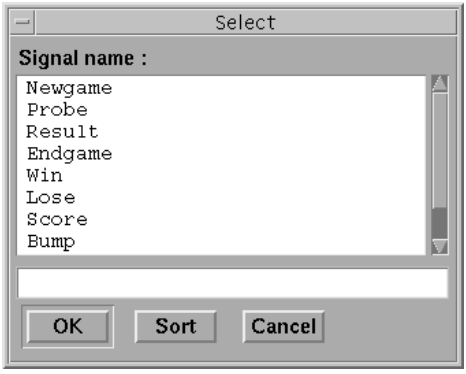


Figure 462: A typical selection dialog

In this dialog, the value can also be entered or edited on the text input line below the list in the dialog. The *Sort* button sorts all values in alphabetical order.

Name completion with the space character is provided. When you press `<Space>` after an initial string, the first value **starting with** the string will be selected (if any). Another `<Space>` will select the next value, etc. When there are no more matches, a space character will be added after the string you initially entered.

A slightly different name completion is provided with the '?' character. When you press '?' after an initial string, the first value **containing** the string will be selected (if any). Another '?' will select the next value. When there are no more matches, a '?' character will be added after the string you initially entered.

For those commands that take an optional variable component, the component must be entered on the text input line after the selected variable name, since it will not be asked for in a dialog. If a '?' is entered instead of a variable component, an additional dialog is opened in which the component can be selected. See the discussion of '?' in "Examine-Variable" on page 2152.

Signal Parameter Dialogs

If a command takes an output signal as parameter, and that signal have parameters, the signal parameters are asked for in a separate dialog. In this dialog, all parameters are listed with their default (“null”) values, and they can all be edited in the same dialog. For more information, see “Selecting Signal Parameters” on page 2246 in chapter 50, *Simulating a System*.

Quick Buttons

The simulator has the following quick buttons:

- **Open simulator**: Open an existing simulator executable file.
- **Restart simulator**: Restart the current simulator from the initial system state.
- **Rerun script**: Rerun last executed script.
- **Undo command(s)**: Undo one or more commands.
- **Show Organizer**: Show the Organizer main window.
- **Show Help**: Show help on Simulator user interface.

The Button Area

The *button area* is used for entering monitor or SimUI commands by clicking the left mouse button on a command button. Each button corresponds to a specific command. The buttons are divided into groups, roughly corresponding to the different types of commands listed in chapter 50, *Simulating a System*. Each group is shown as a *module* in the button area. Any number of button modules may reside in the button area. If the modules do not fit in the button area, a vertical scroll bar is added.

The definition of the buttons and button groups are stored in a *button definition file* (see “Definition Files” on page 2221). New buttons can be added and existing ones deleted or redefined by using the *Group* menu in a button module.

To examine a button’s definition without executing the command, the left mouse button is pressed on the button and the mouse pointer is

moved outside the button. The command definition then appears in the status bar, but the command is not executed.

If a button's definition contains parameters, the parameter values are prompted for in dialog boxes before the command is executed, in the same way as described for commands entered from the input line. See "Parameter Dialogs" on page 2201.

When no more parameter values are requested, the string shown on the input line saved in the history list and is moved to the text area. The command is then executed.

A Button Module

A *button module* looks like this:

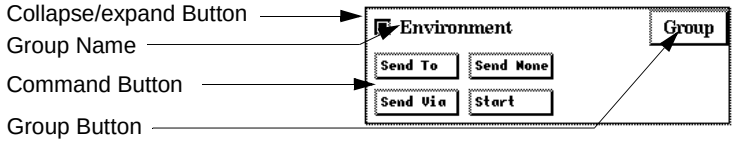


Figure 463: A button module

Each module consists of a title bar and a number of command buttons arranged in rows and columns. The title bar displays:

- A *collapse/expand* toggle button. Clicking on this button collapses the module so that only the title bar is visible, and expands the module back to its current size so that the buttons become visible.
- The *group name* of the button module.
- A *Group* button, providing a menu with commands affecting the buttons in the module.

The *Group* button contains the following menu items:

Add

This menu choice adds a new button to the button module. A dialog prompts for the button label and the command to be executed when the button is pressed. The new button is added to the end of the module. Several buttons may be added with the dialog by using the *Apply* button instead of the *OK* button.

For the syntax of a button definition, see the subsection "Button and Menu Definitions" on page 2222.

> **Note:**
>
> **On UNIX**, if several buttons have **the same button label**, it is always **the first button** found that will be deleted or modified, independently of the selection.

Edit

This menu choice edits the label and definition of a button. The button to edit is selected in a dialog. When a button has been selected, the label and definition can be edited using a dialog.

> **Note:**
>
> **On UNIX**, if several buttons have **the same button label**, it is always **the first button** found that will be edited, independently of the selection.

Delete

This menu choice deletes one or more buttons from the button module. The buttons to be deleted are selected in a dialog.

> **Note:**
>
> **On UNIX**, if several buttons have **the same button label**, it is always **the first button** found that will be deleted, independently of the selection.

Rename Group

This menu choice edits the name of the current button module in a dialog.

Delete Group

This menu choice deletes the current module from the button area. A dialog asks for confirmation.

The Default Button Modules

The following tables list the default buttons in the button modules and the corresponding monitor command. See "Monitor Commands" on page 2145 for more information.

> **Note:**
>
> The buttons in the button modules are specified in the button definition file. If the default button file is not used, the button modules may be different than described here. See "Button and Menu Definitions" on page 2222 for more information.

The *Execute* Module

Button	Monitor command
Symbol	Step-Symbol
Transition	Next-Transition
Go	Go
Until Timer	Proceed-To-Timer
Break	Pressing `<Return>`

The *Send Signal* Module

Button	Monitor command
Send To	Output-To
Send Via	Output-Via
Send None	Output-None

The *Examine* Module

Button	Monitor command
Process List	List-Process -
Variable	Examine-Variable (
Var In Scope	Examine-Variable

The *Trace* Module

Button	Monitor command
MSC	Start-Interactive-MSC-Log 1
SDL	Set-GR-Trace 1
TEXT	Set-Trace 6

The Menu Bar

This section describes the menu bar of the SimUI's main window and all the available menu choices. However, the *Help* menu is described in "Help Menu" on page 15 in chapter 1, *User Interface and Basic Operations*. Simulator commands are described in "Monitor Commands" on page 2145.

The menu bar contains the following menus:

- *File Menu*
- *View Menu*
- *Buttons Menu*
- *Log Menu*
- *General Menu*
- *Execute Menu*
- *Examine Menu*
- *Change Menu*
- *Show Menu*
- *Trace Menu*
- *Breakpoint Menu*
- *Help Menu*
  (See "Help Menu" on page 15 in chapter 1, *User Interface and Basic Operations*.)

File Menu

The File menu contains the following menu choices:

- *Open*
  (See "Open" on page 9 in chapter 1, *User Interface and Basic Operations*.)
- *Restart*
- *Exit*
  (See "Exit" on page 15 in chapter 1, *User Interface and Basic Operations*.)

Restart

Restart the currently opened simulator. After user confirmation, the currently running simulator is stopped, the *Watch* window is updated, and the text area is cleared from previously executed commands.

The command is dimmed if no simulator has been opened.

View Menu

The *View* menu contains the following menu choices:

- *Watch Window*
- *Command Window*
- *Open All*
- *Close All*
- *Clear Text Area*

Watch Window

Opens the Watch window displaying variable values in the simulation. If this window is already opened, the menu item is dimmed. See "Watch Window" on page 2219 for more information about this window.

Command Window

Opens the Command window in which arbitrary monitor commands will be executed. If this window is already opened, the menu item is dimmed. See "Command Window" on page 2216 for more information about this window.

Open All

Opens the Watch and Command windows.

Close All

Closes the Watch and Command windows.

Clear Text Area

Clears the text area in the main window without affecting the simulation.

Buttons Menu

The Buttons menu contains the following menu choices:

- *Load*
- *Append*
- *Save*
- *Save As*
- *Expand Groups*
- *Collapse Groups*
- *Add Group*

For more information on the SimUI's button definition file mentioned in the menu commands below, see "Definition Files" on page 2221.

Load

Reads in a new button definition file that overrides the current button definitions. All buttons and modules currently in the button area are deleted and replaced by the buttons and modules defined in the new file. A File Selection Dialog is opened for specifying the file to load.

Append

Appends the contents of a new button definition file into the current button definitions. Buttons with new labels are added to the button area, while buttons with already existing labels in the same module will be duplicated (possibly with different definitions). A File Selection Dialog is opened for specifying the file to append.

Save

Saves the current button and module definitions in the button definition file under its current file name.

Save As

Saves the current button and module definitions in a new button definition file. A File Selection Dialog is opened for specifying the new file name.

Expand Groups

Expands all modules in the button area.

Collapse Groups

Collapses all modules in the button area.

Add Group

Adds one or more new button module after the last module in the button area. A dialog box is opened for specifying the name of the new module. Several modules may be added with the dialog by using the *Apply* button instead of the *OK* button.

Log Menu

The *Log* menu manages three different log variants:

- The *input history* contains all textual commands sent to the SimUI.
- The *command history* contains all textual commands sent to the simulator.
- The *complete log* contains all textual output presented during a simulator session, this includes both user commands and simulator output.

The difference between the input history and the command history is that the SimUI processes commands before forwarding them to the simulator. The input history contains the raw user input, while the command history contains pure simulator commands. For instance:

- *Macros*: The input history contains

```
Output-to MySignal ($maxvalue) Main
```

while the command history contains

```
Output-to MySignal (37329) Main
```

- *Check*: This SimUI command, together with its expected output section, is saved in the input history but not in the command history.
- *Execute input script*: This SimUI command is saved in the input history, while the command history contains the executed commands.

The *Log* menu contains the following menu choices:

- *Save Input History*
- *Save Command History*
- *Clear Input History*
- *Clear Command History*
- *Start/Stop Complete Log*
- *Log Status*

Clear Input History

SimUI command: `clear-input-history`

Normally, the input history is cleared when the simulator is restarted. You use this menu choice to clear the input history without restarting the simulator.

Save Input History

SimUI command: `save-input-history`

A file selection dialog appears, where you specify the file to save the input history in. The default file extension for input history files, i.e. input scripts, is `*.cui`.

Clear Command History

SimUI command: `clear-command-history`

Normally, the command history is cleared when the simulator is restarted. You use this menu choice to clear the command history without restarting the simulator.

Save Command History

SimUI command: `save-command-history`

A File Selection Dialog appears, where you specify the file to save the command history in. The default file extension for command history files, i.e. command scripts, is `*.com`.

Start/Stop Complete Log

Start Complete Log starts the logging of complete monitor interaction, i.e. the complete contents of the text area will be logged to a file. A file selection dialog is opened for specifying the log file. If an already existing log file is selected, the user is asked whether to overwrite this file or to append the log to it.

Stop Complete Log stops the complete logging to the file. The appropriate menu choice is displayed depending upon the current state of the log.

Log Status

Displays the status of the command and complete logs in a dialog box. If a log is active, the name of the log file is shown.

Additional Simulator Menus

In addition to the standard SimUI menus, a few special simulator menus are included in the menu bar. The menu choices in these menus simply execute a monitor command, i.e. they are functionally equivalent to buttons in the button modules. If the monitor command requires parameters, they are prompted for using dialogs in the same way as the command buttons.

The following tables list the default menu choices and the corresponding monitor command. See "Monitor Commands" on page 2145 for more information.

> **Note:**
>
> The additional menus in the SimUI are specified in the button definition file. If the default button file is not used, the button modules may be different than described here. See "Button and Menu Definitions" on page 2222 for more information.

General Menu

Menu choice	Monitor command
Command	? (Interactive Context Sensitive Help)
Start SDL Env	Start-SDL-Env
Version	Show-Versions
News	News

Execute Menu

Menu choice	Monitor command
Go	Go
Over Symbol	Next-Symbol
Into Stmt	Step-Statement
Over Stmt	Next-Statement
Finish	Finish
Until Time	Proceed-Until
Until Trace	Next-Visible-Transition
Until Timer	Proceed-To-Timer
Input Script	`execute-input-script`
Command Script	Include-File
Stop Sim	Exit

Examine Menu

Menu choice	Monitor command
Ready Q	List-Ready-Queue
Now	Now
Process List	List-Process
Input Port	List-Input-Port

Menu choice	Monitor command
Signal	Examine-Signal-Instance
Timer List	List-Timer
Variable	Examine-Variable (
Call Stack	Stack
Set Scope	Set-Scope

Change Menu

Menu choice	Monitor command
Ready Q	Rearrange-Ready-Queue
State	Nextstate
Create Process	Create
Stop Process	Stop
Input Port	Rearrange-Input-Port
Del Signal	Remove-Signal-Instance
Set Timer	Set-Timer
Reset Timer	Reset-Timer
Variable	Assign-Value
Synonym File	`set-synonym-file <file name>`

Show Menu

Menu choice	Monitor command
Next Symbol	Show-Next-Symbol
Prev Symbol	Show-Previous-Symbol
C Line	Show-C-Line-Number
Coverage	Show-Coverage-Viewer

Trace Menu

Menu choice	Monitor command
Text Level : Set	Set-Trace
- : Show	List-Trace-Values
SDL Level : Set	Set-GR-Trace
- : Show	List-GR-Trace-Values
MSC Level : Set	Set-MSC-Trace
- : Show	List-MSC-Trace-Values
MSC Trace : Start	Start-Interactive-MSC-Log
- : Start Batch	Start-Batch-MSC-Log
- : Stop	Stop-MSC-Log

Breakpoint Menu

Menu choice	Monitor command
Transition	Breakpoint-Transition
Output	Breakpoint-Output
Variable	Breakpoint-Variable
Symbol	Breakpoint-At
Connect sdle	Connect-To-Editor
Remove	Remove-All-Breakpoints
Show	List-Breakpoints

Command Window

The *Command window* is an optionally visible window used for displaying the results of executed commands. The Command window is opened from the SimUI's *View* menu.

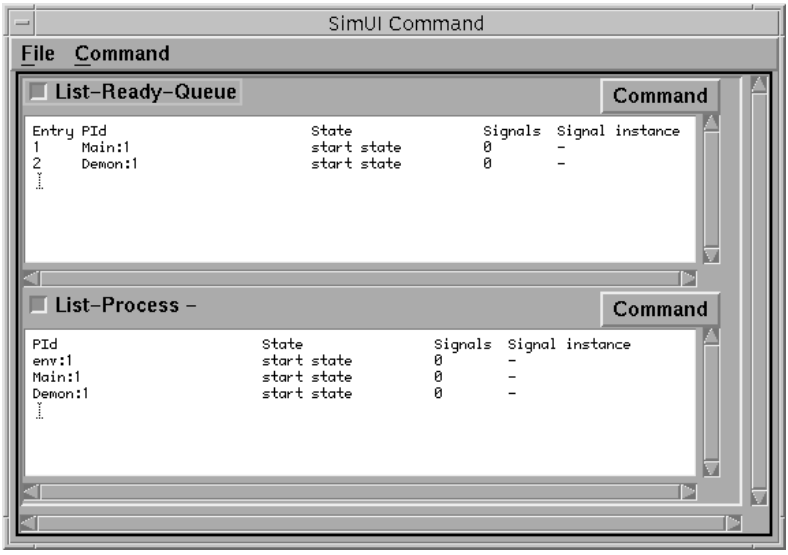


Figure 464: The Command window

The Command window is updated automatically whenever the monitor becomes active and after each monitor command. See "Activating the Monitor" on page 2131 for information on when the monitor becomes active. The window can also be updated manually with a menu command.

Any number of commands can be defined to be executed in the Command window. Each command is executed in a scrollable module in the window.

New commands can be added to the window and existing commands can be changed. The commands to execute are by default List-Process and List-Ready-Queue. The set of commands to execute are stored in a *command definition file* (see "Definition Files" on page 2221). The default command definition file can be changed with the Preference Manager.

Command Modules

A *command module* looks like this:

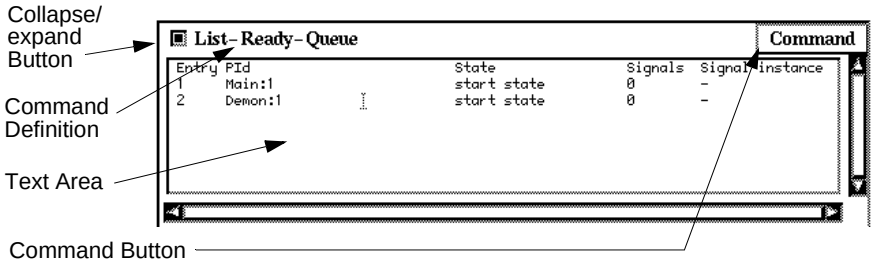


Figure 465: A command module

Each module consists of a *title bar* and a scrollable *text area* for the command output. The title bar displays:

- A *collapse/expand* toggle button. Clicking this button collapses the module so that only the title bar is visible, and expands the module back to its current size so that the text area becomes visible. When a module is expanded, the text area is automatically updated.
- The *command* that is executed.
- A *Command* button, providing a menu with commands affecting the module.

The *Command* button contains the following menu items:

Edit

Opens a dialog for editing the command executed.

Delete

Deletes the command and command module from the Command window.

Size

Sets the size of the text area. A dialog is opened where the number of text rows can be set using a slider.

File Menu

The File menu contains the following menu choices:

- *Load*
- *Append*
- *Save*
  (See "Save" on page 11 in chapter 1, *User Interface and Basic Operations*.)
- *Save As*
  (See "Save As" on page 12 in chapter 1, *User Interface and Basic Operations*.)
- *Close*
  (See "Close" on page 14 in chapter 1, *User Interface and Basic Operations*.)

Load

Reads in a new command definition file that overrides the current command definitions. All commands currently in the Command window are deleted and replaced by the commands defined in the new file. A File Selection Dialog is opened for specifying the file to load.

Append

Appends the contents of a new command definition file into the current command definitions. New commands are added to the command window, but already existing commands are not affected. A File Selection Dialog is opened for specifying the file to append.

> **Note:**
>
> Command names in a command definition file are **case sensitive**. If a command in an appended file already exists in the Command window, but with different case, the command will be duplicated.

Command Menu

The Command menu contains the following menu choices:

Command

Adds a new command to the Command window. A dialog box prompts for the new command. The new command module is added to the bottom of the window. For the syntax of a command definition, see "Command Definitions" on page 2223.

Update All

Updates all command modules by executing the defined commands.

Watch Window

The *Watch window* is an optionally visible window used for displaying values of variables defined in the currently running simulation. The Watch window is opened from the SimUI's *View* menu.

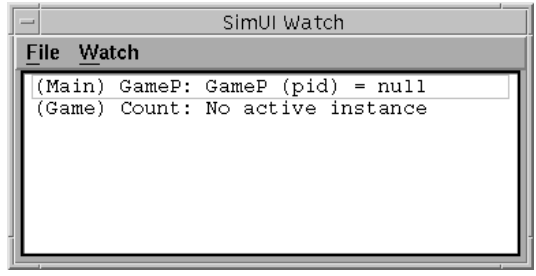


Figure 466: The Watch window

The Watch window is updated automatically after each monitor command. The window can also be updated manually with a menu command.

The variables to display are selected with a menu command. The set of selected variables are stored in a *variable definition file* (see "Definition Files" on page 2221).

File Menu

The File menu contains the following menu choices:

- *Load*
- *Append*
- *Save*
  (See "Save" on page 11 in chapter 1, *User Interface and Basic Operations*.)
- *Save As*
  (See "Save As" on page 12 in chapter 1, *User Interface and Basic Operations*.)
- *Close*
  (See "Close" on page 14 in chapter 1, *User Interface and Basic Operations*.)

Load

Reads in a new variable definition file that overrides the current variable definitions. All variables currently in the Watch window are deleted and replaced by the variables defined in the new file. A File Selection Dialog is opened for specifying the file to load.

Append

Appends new variable definitions to the current variable definitions file. The variables in the file are added to the Watch window. A File Selection Dialog is opened for specifying the file to merge. Duplicate variable definitions are permissible.

Watch Menu

The Watch menu contains the following menu choices:

- *Update All*
- *Add*
- *Edit*
- *Delete*
- *Delete All*

Update All

Updates the Watch window by showing the current value of all displayed variables.

Add

Adds one or more variables to the Watch window. A dialog is opened for specifying the variable to be added. For the syntax of a variable definition, see "Variable Definitions" on page 2224. Several modules may be added with the dialog by using the *Apply* button instead of the *OK* button.

Edit

Edits a variable specification in the Watch window. The variable whose specification is to be changed is selected in a dialog. When a variable has been selected, the specification can be edited using a dialog.

Delete

Deletes one or more variables from the Watch window. The variables to delete are selected in a dialog.

Delete All

Deletes all variables from the Watch window.

Definition Files

In the SimUI, the following types of information are stored on files:

- **Button definitions**

  i.e. definitions of button groups and button commands in the main window's button area.

- **Menu definitions**

  i.e. definitions of additional menus and menu commands in the main window's menu bar.

- **Command definitions**

  i.e. definitions of commands to be executed in the Command window.

- **Variable definitions**

  i.e. definitions of variables to display in the Watch window.

At start-up of the SimUI, the files to read are determined in the following way:

1. The file names are defined with the Preference Manager. If a file name is not defined there, the default file name `def.btns`, `def.cmds` and `def.vars` is used, respectively.

2. If the file names does not contain a directory path, the files are searched for in the following directories and the following order:

   – the current directory
   – the user's home directory
   – the installation directory

Once a file has been found, it is read and the contents of the corresponding window are set up. If a file cannot be found, the corresponding window area becomes empty.

Common File Syntax

Each of three text files can contain comment lines starting with the '#' character. Empty lines are not discarded.

> **Note:**
> When a file is read, **no checks are made upon the relevance** or correctness of the definitions contained in the file.

Button and Menu Definitions

The *button and menu definitions* are stored in a button definition file with the default extension `.btns`. The button definitions are divided into groups where each group defines a button module in the main window's button area, or a menu in the main window's menu bar.

Syntax

In the file, a button group has the following syntax:

```
:[COLLAPSED] <group name>
<button label>
<definition>
<button label>
<definition>
  . . .
```

The <group name> is the string shown in the title bar of the button module. If the group name is prefixed with the string `COLLAPSED`, the button module is initially collapsed. The <button label> is the label of the button in the button module. The <definition> is the monitor command that will be executed when the button is pressed. The syntax of a button definition is the same as when entering a command textually to the monitor.

A menu has the following syntax:

```
:MENU <menu name>
<menu choice>
<definition>
<menu choice>
<definition>
 . . .
```

The <menu name> is the name of the menu shown in the menu bar. The <menu choice> is the name of the menu choice in the menu. The <definition> is the monitor command that will be executed when the menu choice is selected.

In the monitor command definition, a '?' as parameter will not work, but a hyphen '-' can be used to signify the default value. Missing parameters at the end of the command will open dialogs for those parameters.

Defining Multiple Commands

In addition, a button definition may consist of several monitor commands, if they are separated by a space and a semicolon, i.e. ";". The commands are then executed immediately after each other.

Example 336: Multiple Commands in Simulator ––––––––––––––––––

```
next-transition ; out-via probe -
```

This button definition executes the next transition and then sends the signal Probe from the environment.

––

Command Definitions

The command definitions are stored in a command definition file with the default extension `.cmds`. Each command definition defines a command module in the Command window. The file has the following syntax:

```
<command>
<command>
   . . .
```

The <command> is the monitor command that will be executed in the command module. The syntax of a command definition is the same as when entering a command textually to the monitor. All parameter values must be specified explicitly or with the default value '-', i.e. '?' is not allowed, and no parameters may be missing. Command names are case sensitive.

> **Note:**
>
> Care should be taken when deciding what command to execute; commands belonging to the Execute group **should be avoided**.

Variable Definitions

The variable definitions are stored in a variable definition file with the default extension `.vars`. Each variable definition defines a variable to display in the Watch window. The file has the following syntax:

```
(<process>) <variable>
(<process>) <variable>
   . . .
```

The <process> is the name of the process, possibly augmented with an instance number separated with a space or colon. The <variable> is the name of the variable in that process. Names are **not** case sensitive.

Example 337: Variable Definitions in Simulator

```
(Main) Count
(game:1) guess
```

Macros

The SimUI has a built-in macro facility. A macro can represent a simulator command or a part of a simulator command. Even though SimUI macro commands are processed by the SimUI, it is possible to enter SimUI macro commands almost as if they were normal simulator commands:

- Type in the SimUI macro command. Note though, that SimUI commands cannot be abbreviated in the same way as simulator commands.
- Define a button or a menu for the SimUI macro command.

These commands are available for macros:

- Use **add-macro** to define a new macro:

  ```
  add-macro <macro name> <macro value>
  ```

  Example:

  ```
  add-macro maxvalue 37329
  ```

- Use **list-macros** to see all defined macros and their values:

  ```
  list-macros
  ```

  Example:

  ```
  Command: list-macros
  Number of macros defined: 1
  Macro name: maxvalue
  Macro value: 37329
  ```

- Use **$** to get the value of a defined macro:

  ```
  $<macro name>
  ```

  Example: The SimUI input:

  ```
  Output-to MySignal ($maxvalue) Main
  ```

  is sent to the simulator as

  ```
  Output-to MySignal (37329) Main
  ```

- Use **remove-macro** to remove an already defined macro:

  ```
  remove-macro <macro name>
  ```

  Example:

  ```
  remove-macro maxvalue
  ```

SimUI Commands

The SimUI examines entered textual commands before sending them to the simulator. If they are SimUI commands they are not sent to the simulator.

Here is a list of all SimUI commands and what they do:

- `add-macro`

  Adds a new SimUI macro. For details, see "Macros" on page 2224.

- `check`

  Tells the SimUI that the textual simulator output from the next command should be checked.

  – When commands are entered manually and a check command is issued, the SimUI saves the simulator output for the next command in an expected output section in the input history.

  – When commands are read from file by executing an input script, and a check command is encountered, the SimUI compares the simulator output for the next command with the expected output from the input script. The SimUI reports if the check passed or failed.

- `clear-command-history`

  Clears the simulator command history without restarting the simulator. This command is useful if you want to save a command history that does not start from the beginning of a simulator session.

- `clear-input-history`

  Clears the SimUI input history without restarting the simulator. This command is useful if you want to save an input history that does not start from the beginning of a simulator session.

- `execute-input-script`

  Executes the commands found in an input script, i.e. an input history saved to file.

  – If a relative file name is used, then it is interpreted as relative to the Organizer source directory.

  – If parameters are given as a space separated list after the file name, then these parameters can be accessed with $1, $2, ... in the called script. It is also possible to access the number of parameters with $# and the name of the calling script with $0.

- `list-macros`

Produces a list of all defined macros and their values. For more details, see "Macros" on page 2224.

- `remove-macro`

  Removes an already existing SimUI macro. For more details, see "Macros" on page 2224.

- `save-command-history <file name>`

  Saves the simulator command history in a command script file (`*.com`). If no <file name> is given, a file selection dialog appears, where you can specify a file name.

- `save-input-history`

  Saves the SimUI input history in an input script file (`*.cui`). If no <file name> is given, a file selection dialog appears, where you can specify a file name.

- `set-source`

  Saves an SDT reference so it is possible to navigate to the symbol in the MSC diagram that the following commands originated from. It is not intended for a user to enter, instead it is automatically generated by the MSC translator.

- `set-synonym-file <file name>`

  Sets the synonym file to use the next time the simulator is restarted from the simulator UI. If no <file name> is given, a file selection dialog appears, where you can specify a file name. Note that the synonym file can also be set from the menu choice SimUI>Change>Synonym File.

Regression Testing

By using the simulator for regression testing, it is possible to check that an enhanced version of an SDL system does not break the old functionality, that is still expected to work for the same SDL system.

With regression testing in the simulator, you can for instance:

- Send in signals and check that the correct signals are sent back out again.
- Check that a variable has the correct value at a specific moment.

The basic idea is to save and replay the input history, and check the textual simulator output for selected commands. An input history saved to a file is called an input script. A group of input scripts can be collected in the Organizer together with the SDL system. The Organizer provides the possibility to run a group of input scripts (or test cases). This results in a textual summary where it is easy to see any test case failures. To examine a test case failure in detail, it is possible to run a single test case with the SimUI command `execute-input-script` (see "SimUI Commands" on page 2225).

Note that it is also possible to express test cases with the MSC notation. MSC test cases are converted to input scripts before being run in the simulator.

Here is a small example, to see how it works:

Example 338 ----------

In System DemonGame (that is provided as an example in the SDL Suite installation), you can (after some initialization) send in a Result signal and expect to get a Score signal back.

To make a test case testing that we get a Score signal back:

1. Create a simulator for System DemonGame.
2. Execute the following simulator commands (you do not have to type in the comments):

```
- Output-To NewGame Main /* Send in signal NewGame
  to start a new game. */
- Next-Transition /* Execute the start transition
  for process Main. */
- Next-Transition /* Execute the start transition
  for process Demon. */
- Next-Transition /* Execute the transition receiv-
  ing signal NewGame and creating process Game */
- Output-To Result Game /* Send in a signal to get
  the current score. */
- break-output - env - - - - - - /* Set a breakpoint
  on signals sent to the environment. */
- Set-Trace System DemonGame 0 /* Minimize textual
  trace. */
- check /* Check the output from the next command. */
- go /* Execute as far as possible. The simulator
  will stop and report that the Score signal has been
  sent to the environment. */
```

3. The go command that we are checking produces the following textual simulator output:
   ```
   Breakpoint matched by output of Score
   ```
4. Save the input history in an input script, with the SimUI command `save-input-history`. The last part of the input script will look like this:

   ```
   check
   go
   expected-output start
   Breakpoint matched by output of Score
   expected-output end
   ```

5. Add a symbol for the input script in the Organizer, with *Edit > Add New*, plain text, do not show in editor.
6. Connect the symbol to the input script file with *Edit > Connect*.
7. Now, make a test: Deselect everything in the Organizer (to make sure that both the SDL system and the input script will be considered) and choose *Tools > Simulator Test*. After a dialog, a fresh simulator will be created and the test case run. The result is presented in the Organizer Log.
8. To force a test case failure:
   - Edit the test case (by double-clicking on the Organizer symbol to get a text editor) by changing the expected output to be something different.
   - Save the test case.
   - Repeat the above test.
   - If everything works as expected, the test case fails.
9. To examine the details of a test case failure:
   - Restart the simulator.
   - Execute the test case (input script) in the SimUI with the `execute-input-script` command.
   - If everything works as expected, the expected output from the go command did not match the actual output.

Mapping Instances to Different Environments

When generating test scripts (.CUI scripts) from MSCs, it is possible to select for which specific processes/blocks for which a cui script shall be generated if an MSC testcase describes several blocks.

Through an instance translation table, which is a plain text file in the Organizer with the extension `*.itt`, it is decided which instances in the MSC that should be regarded as environment. In this file it is for example stated that an instance A really is instance `env:2`.

```
Example *.itt file:
MyBlock env:2
MyProcess env:3
```

The name used for an instance in the MSC is placed first, followed by a space, and then comes the name of the environment instance to map the MSC instance to. Note that the only names that can be used for mapping are `env:2`, `env:3`, `env:4` etc. and that no numbers can be left out when mapping, i.e. it is not allowed to only use `env:2` and `env:4` without using `env:3`.

Whenever an MSC is converted to a CUI test script, the first found (or closest found) translation table from the Organizer is used and all instances in the file are translated according to the table in the file. All instances starting with `env` (after translation, if there is a translation file) are outside the system, and all other instances are assumed to be inside the system.

Order of execution

In the case where the MSCs used as test scripts only have one environment instance, the order of execution is the same order as messages are connected to the environment instance in.

To determine the order things are done in when several instances are converted to environment instances, the y-coordinate of the connection of messages to environment instances is used. If the y-coordinate of sending in message A is above the y-coordinate of sending in message B, then message A is sent before message B. If two connections have the same y-coordinate, then the x-coordinate is used to determine the order. The left most connection is done first.

Only messages connected to an environment instance in one end and a system instance in the other end are used to drive the simulation. Mes-

sages between environment instances and messages within the system are ignored.

For action symbols (that can contain ordinary simulator commands or shortcuts for ordinary simulator commands), the upper connection point to the environment instance is used for determining the order of doing things in.

It is also possible to test in an MSC test script if a service response message is returned to the same environment instance (representing a specific external process) that sent in the service request message, in the case where there are many environment instances. This is done with the *output-internal* simulator command that is generated by the MSC to CUI converter. An example:

```
output-internal SignalIn ProcessOne env:2
```

If the system responds to this message by sending back another message to sender, then the textual trace will contain information about what environment instance the signal was sent to:

```
* OUTPUT of SignalOut to env:2 from ProcessOne:1
```

In addition to the output-internal command, the create null env command is generated in the CUI script by the converter. This command will create the environments in the simulator that will be used for the instances that are mapped to an environment instance in the `.itt` file.

Note that when instances are mapped with an `.itt` file, process id mapping should not be used.

Restrictions

Restrictions on Monitor Input

The following restrictions apply to the monitor input:

- A parameter to a monitor command may not contain more than 5,000 characters.
- **On UNIX**, control characters of different types may terminate the simulation program. `<Ctrl+C>`, `<Ctrl+D>`, and `<Ctrl+Z>` are typical characters that might terminate a simulation program.

Restrictions on Dynamic Checks

There are a number of dynamic checks that are not performed at all or performed at the C level by the C runtime system. A C runtime error will of course lead to the simulation program being terminated. The following checks are not made at the SDL level.

- Several paths from a decision for a certain decision expression value. A simulation program will simply choose one of the existing paths.
- Overflow of integer and real values are checked at the C level if the actual C system performs these checks. (These checks are likely not to be performed.)

Chapter

50 *Simulating a System*

This chapters provides general information related to simulation in the SDL Suite and describes the actions you perform when simulating an SDL system.

For a reference to the simulator user interface, see *chapter 49, The SDL Simulator*.

Structure of a Simulator

In order to generate a simulator, the SDL to C Compiler is used. The simulators generated using the SDL to C Compiler consist of the following components:

- The Simulated System (the application itself)
- The Environment Process
- The Interactive Monitor System
- The Graphical User Interface (the Simulator UI).

Additional tools support graphical trace when executing a simulator:

- The SDL Editor
- The MSC Editor.

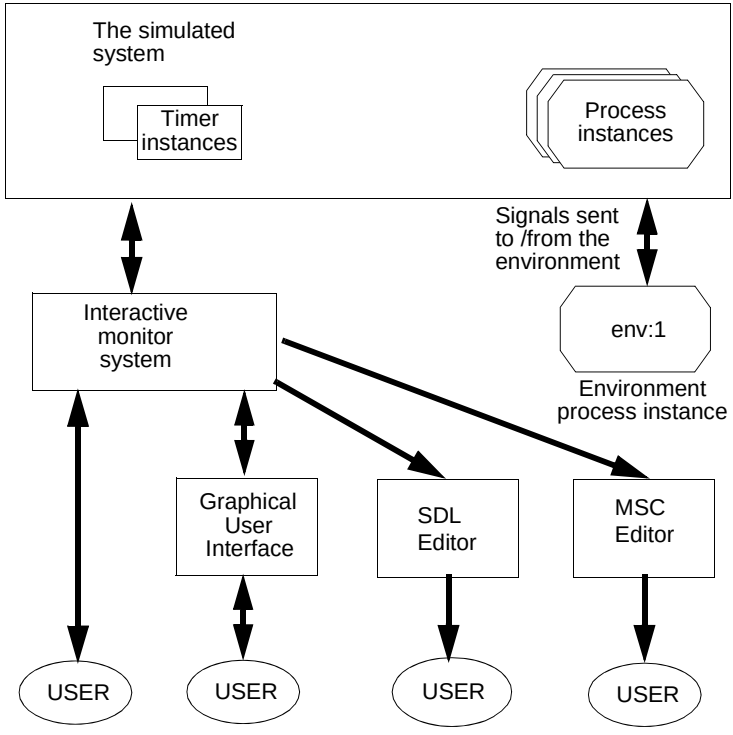


Figure 467: Structure of a running simulator

The Simulated System

The application is a representation in C code of the implementation of the SDL system. Having generated the application (that is, translated the SDL system into C code), the SDL to C compiler generates a make-file which contains instructions about how to compile and link the application together with the appropriate runtime library. A number of libraries are provided, which allow you to generate an application with the desired behavior. For instance, it is possible to have multiple simulators communicate with each other or communicate with an external application, provided that application has the facility to communicate through the Postmaster.

An executing simulation program contains a number of data objects that represent certain SDL objects in the system that is being simulated. Process instances, signal instances (signals that are waiting in the input port of a process instance), and timer instances (timers that are set, but that have not output their corresponding signal instance) are all examples of such objects. These objects, together with the process instance env:1, which represents the environment of the system, and the monitor system, constitute the simulation program (see Figure 467).

The process instances in the simulated system will execute transitions that consist of actions like tasks, decisions, outputs, procedure calls, and so on, according to the rules of SDL. It is assumed that a transition takes no time and that a signal instance is immediately placed in the input port of the receiver when an output operation occurs.

The Environment Process

The function of the environment process instance, env:1, is to be a receiver of signal instances sent from the system to the environment of the system. The input port of env:1 will always hold the last 20 signals sent to the environment.

The Interactive Monitor System

When generating a simulating application, a *monitor system* is included, the function of which is to allow the interaction between the application and the user via a command line user interface. The monitor system provides a functionality similar to high-level language debuggers, in which you can, for instance:

- view and modify objects defined in your SDL system,
- set breakpoints,
- control the execution,
- control the logging facilities,
- and so on...

The interactive monitor is the interface between the user and the simulated system. The monitor can be seen as an ordinary process instance, which, when executing a transition, accepts commands from the user.

One group of monitor commands terminate the current transition of the monitor, allow one or several other process instances to execute their transitions, and then start a new transition by the monitor.

The Graphical User Interface

A graphical user interface to the monitor system of a simulator, known as the *Simulator UI*, is also provided. It is a window-based tool which facilitates the use of the monitor system. It contains graphical components, such as buttons, menus, and scrollable lists, which make it easier to run the simulator in an intuitive way. Also, it is designed to minimize the required amount of user interaction.

> **Note:**
>
> Running the graphical user interface requires that the Cbasic SDL to C Compiler is present in your configuration.

When running the Simulator UI, the user has the option to take advantage of the graphical features which are provided, or to enter commands in a command line fashion in the same way as the textual interface.

Generating and Starting a Simulator

There are two ways to generate and start a simulator:

- A quick way in one single step, adequate for most situations
- A more complex way in several steps, giving you complete control of the generation and start process.

In the following, the more complex way will be described first, to give a full understanding of the process. The quick way is described in "Quick Start of a Simulator" on page 2239.

Generating a Simulator

A Simulator for an SDL system, or a part of an SDL system, consists of the C code generated by the SDL to C compiler together with a predefined run-time kernel. To start a Simulator, it is thus necessary to first generate an executable simulator. This is performed in the Organizer.

To generate an executable simulator:

1. Select a system, block, or process diagram in the Organizer.
2. Select *Make* from the *Generate* menu. The Make dialog is opened.
3. Turn on the options *Analyze & generate code* and *Makefile*.
4. From the *Standard kernel* option menu, select *Simulation*.
5. If you need to check the Analyzer options, click the *Analyze Options* button. In the dialog, set the options and click the *Set* button. For more information about these options, see "Analyzing Using Customized Options" on page 2618 in chapter 55, *Analyzing a System*.
6. Click the *Make* button.

   On UNIX, a simulator for the system is now generated in the current directory with the name `<system>_xxx.sct` (the `_xxx` suffix is platform specific).

   In Windows, a simulator for the system is now generated in the current directory with the name `<system>_xxx.exe` (the `_xxx` suffix is kernel and compiler specific).

   The Status Bar of the Organizer reports the progress of the generation; the last message should be "Compiler done."
7. Open the Organizer Log window from the *Tools* menu and check that no errors occurred and that a simulator was generated.
   - If errors were found, correct them and repeat the generation process. See "Locating and Correcting Analysis Errors" on page 2624 in chapter 55, *Analyzing a System*.
   - If no simulator was generated, repeat the generation process, but click the *Full Make* button in the Make dialog instead.

Starting a Simulator

An executable simulator can be run in two different modes; graphical mode and stand-alone mode (textual mode).

Graphical Mode

In graphical mode, the Simulator takes advantage of the graphical user interface and integration mechanism of the SDL Suite. A separate graphical user interface, the *Simulator UI*, is started, giving access to the monitor system through the use of menus, command buttons, etc.

To start a simulator in graphical mode:

1. Select *SDL > Simulator UI* from the Organizer's *Tools* menu. The graphical user interface of the Simulator is opened (see "The Graphical Interface" on page 2244).
2. Select *Open* from the Simulator UI's *File* menu. A File Selection Dialog is opened.

   – Alternatively, click the *Open* quick button in the tool bar.

3. In the dialog, select an executable simulator and click *OK*.

A welcome message is printed in the text area of the Simulator UI. The monitor system is now ready to accept commands. Please see "Supplying Values of External Synonyms" on page 2240 for some additional information that may affect the start-up.

Stand-Alone Mode (Textual Mode)

In stand-alone mode, the Simulator uses the input and output devices currently defined on your computer, which provide a textual, command line based user interface. A very limited graphical support is provided when running the Simulator in this mode.

To start a simulator in stand-alone mode, the generated simulator is executed directly from the OS prompt, e.g.

```
csh% ./system_sma.sct
```

A welcome message is printed on the terminal:

```
Welcome to SDL SIMULATOR. Simulating system <system>

Command :
```

The monitor system is now ready to accept commands. Please see "Supplying Values of External Synonyms" on page 2240 for some additional information that may affect the start-up.

> **Note:**
>
> **On UNIX**, before a simulator can be run in stand-alone mode, you must execute a command file from the operating system prompt. The file is called `telelogic.sou` or `telelogic.profile` and is located in the binary directory that is included in your `$path` variable.
> For csh-compatible shells: **`source <bin dir>/telelogic.sou`**
> For sh-compatible shells: **`. <bin dir>/telelogic.profile`**

Quick Start of a Simulator

A simulator can also be generated and automatically started in graphical mode in one single step.

To quick start a simulator, click the *Simulate* quick button in the Organizer's tool bar. The following things happen:

- A simulator is generated by using the simulator kernel that is specified in the Make dialog. (If no simulator kernel is specified, a default simulator kernel is used.)
- The graphical Simulator UI is started. If a Simulator UI with the same simulator name is already open, this UI is reused. If another Simulator UI is open, a dialog is opened where you can select to start a new UI, or to reuse one of the existing UI's.
- The generated simulator is started from the Simulator UI.

Restarting a Simulator

An executing simulator can be restarted from the beginning to reset its state completely:

- In graphical mode, select *Restart* from the Simulator UI's *File* menu. (This is the same as opening the same simulator again.) A confirmation dialog is opened.
- In stand-alone mode, the simulator has to be exited with the Exit command and then executed from the OS prompt again.

Supplying Values of External Synonyms

The SDL system for the simulator may contain external synonyms that do not have a corresponding macro definition (see "External Synonyms" on page 2650 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*). In that case, you will be asked to supply the values of these synonyms, either by selecting a file with synonym definitions, or by entering each synonym value from the keyboard.

In stand-alone mode, the following prompt appears:

```
External synonym file :
```

Enter the name of a file containing synonym definitions, or press `<Return>` to be prompted for each synonym value.

In graphical mode, a file selection dialog is opened. Either select a file (`*.syn`) containing synonym definitions, or press *Cancel* to be prompted for each value in a separate dialog. In this dialog, the name and type of the synonym is shown together with an input text field. You can now do the following:

- Enter a value and click *OK*.
- Click *Default value* to get a "null" value for the synonym type entered in the input field. Accept or edit this value and click *OK*.
- Click *Cancel* to give the synonym a "null" value (without the possibility to edit the value).
- Click *Cancel all* to give the synonym and all following synonyms a "null" value.

If a synonym file is selected in the file selection dialog, this file is also used when the simulation is restarted. (If you want to use another synonym file you have to `set-synonym-file` simulator UI command.)

If you set the environment variable `SDTEXTSYNFILE` to a file before starting the SDL Suite, this file will automatically be used. If `SDTEXTSYNFILE` is set to "[[" all synonyms are given "null" values.

Note that the simulator lists all external synonyms that have been given values in another way than from a synonym file. This is done to make it easy to cut and paste information to a synonym file, for future use. An example:

```
The following external synonyms were not found in
```

```
synonym file
/home/lat/simulator.syn:

myThirdSynonym (integer)
```

The syntax of a synonym file is described in "Reading Values at Program Start up" on page 2651 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*.

Actions on Simulator Start-up

When a simulator is started, the static process instances in the system are created, but their initial transitions are not executed.

If a file called `siminit.com` exists in the current directory an implicit Include-File command will be done on this file at startup.

Issuing Monitor Commands

Whenever the interactive monitor system becomes active, commands are accepted from the user. When running the simulator in stand-alone mode, commands can only be entered textually from a command prompt. When running the simulator in graphical mode, commands may be issued both textually and through the use of menus and buttons.

Activating the Monitor

The simulator's monitor system becomes active when the simulator is started, when the execution has reached a stop condition or a breakpoint, when the system is completely idle, when a semantic error occurs, or when the execution is manually stopped.

These conditions are listed in greater detail in *"Activating the Monitor" on page 2131 in chapter 49, The SDL Simulator*.

The Textual Interface

The commands to the monitor system consist of a command name and possibly one or more parameters to the command, separated by spaces or carriage return. The following general rules apply:

- All command names and parameter values are case insensitive. They may also be abbreviated, as long as they are unique. For in-

stance, the command List-Ready-Queue can be entered in any of the following ways:

```
List-Ready-Queue
list-ready
l-r-q
```

- If not all parameters to a command are given on the command line, the missing parameters will be prompted for. A parameter may be optional, in which case a default value always exists. Default values may also exist for required parameters.
- If you enter an unallowed value for a parameter, an error message is printed and the command is not executed. This is the only way to cancel a command that has been entered.

Getting Command Help

To get a simple list of all possible command names, enter a '?' at the command prompt. A command can then be entered after the list.

To get a list of all commands, grouped into different categories, use the command Help without parameters.

To get a list of all possible commands starting with a certain string, enter the string and hit <Return>. Unless the string was a complete command, all command names starting with the entered string are listed. The following example shows how to list all output commands:

```
Command : out
Command was ambiguous, it might be an abbreviation
of:
Output-To Output-Via Output-None Output-Internal
```

To get short help on a specific command, use the Help command and specify the command in question as a parameter, e.g.

```
Command : help help

Help <Optional command name>

Issuing the help command will print all the
available commands. If help is entered with a
command name, this command will be explained.
```

Entering Parameters

To get a list of possible parameter values, enter a '?' on the command line or when prompted for a missing parameter.

To specify a default value for a parameter, enter a '-' on the command line or when prompted for a missing parameter.

If a default value exists for a parameter, you may also hit `<Return>` when prompted for the parameter to accept the default value. However, if no default value exists, this will list the possible values and prompt for the parameter again.

Hint:

Before you are acquainted with the monitor commands and their parameters, it is generally a good idea to only enter the command names, without parameters. When prompted for parameters, use '?' to list possible values, and '-' to accept default values.

The Graphical Interface

The graphical Simulator UI is illustrated in the figure below:

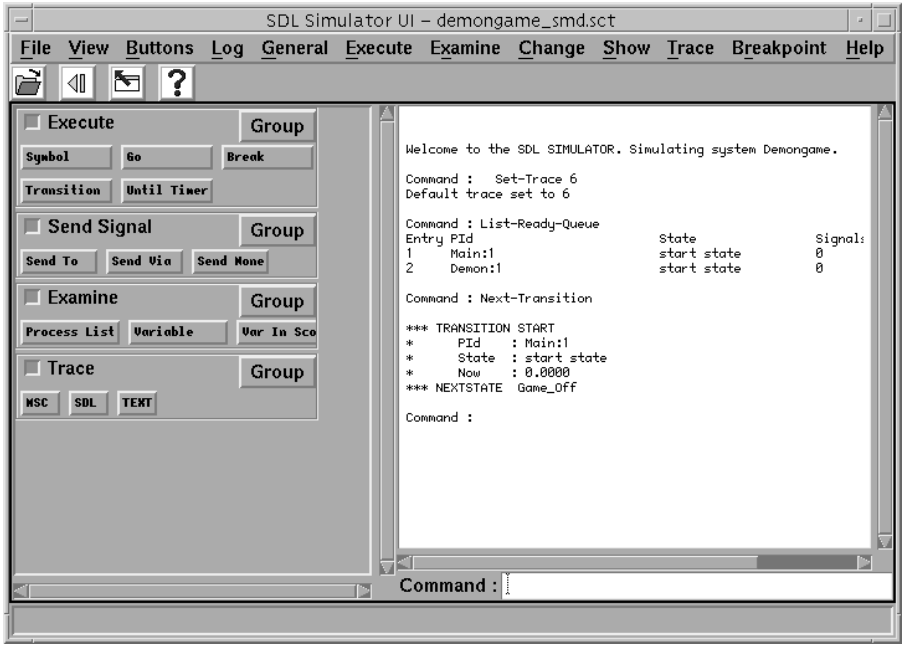


Figure 468: The Simulator UI

All input and output texts to and from the simulator monitor is displayed in the text area to the right.

Entering Commands Textually

To enter monitor commands textually:

1. Place and click the mouse pointer inside the *Command:* input field at the bottom right, below the text area.
2. Enter the command on the input line. The same syntax is used as when running in stand-alone mode; see "The Textual Interface" on page 2241. However, prompting for missing parameters are done by issuing dialogs, where the parameter value can be entered or select-

ed from a list of possible values. See "Selecting Parameters" on page 2245.

Monitor commands entered textually are saved in a *history list*. To re-execute a command in the history list:

1. Place the mouse pointer inside the input field.
2. Use the arrow keys `<Up>` and `<Down>` to display the commands that have been entered earlier.
3. Hit `<Return>` on the command line to re-execute the displayed command. You may edit the command before executing it.

Issuing Commands Using Buttons

The preferred way to issue monitor commands is by using the command buttons in the left area of the Simulator UI, or by using menu choices in the menu bar. Each button or menu choice correspond to a certain monitor command. The buttons are grouped into modules, corresponding to different categories and uses of the commands. The groups are similar to those listed when using the Help command in the textual interface.

To "preview" the command associated with a button or menu choice, without executing it:

1. Place the mouse pointer on the button or menu choice and press the mouse button. The name of the monitor command is displayed in the Status Bar at the very bottom of the Simulator UI.
2. Move the mouse pointer outside the button or menu and release the mouse button.

To execute a command, simply click on a button.

Selecting Parameters

Parameters to monitor commands are prompted for in dialogs (unless they have been specified on the command input line). If a parameter has a set of possible values, the list of values is presented from which you may select a value:

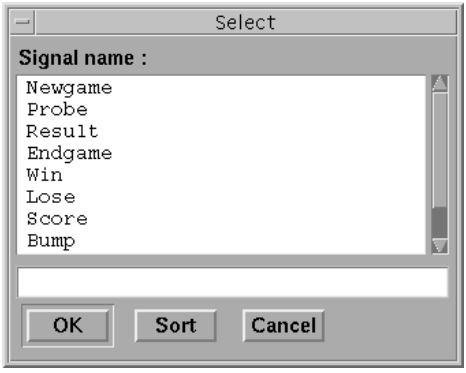


Figure 469: A typical parameter dialog

In this dialog, the values can be sorted alphabetically by clicking the *Sort* button. After having entered an initial string, you can also press `<Space>` to select the first value **starting with** this string, or press '?' to select the first value **containing** this string. Additional `<Space>` or '?' characters will select the next value, etc. When there are no more matches, a real space or '?' character will be added after the string you initially entered.

File parameters are prompted for in File Selection Dialogs. Other parameters, like integers, are prompted for in a simple text input dialog.

- To accept a selected or entered value, click *OK* in the dialog.
- To enter a "null" value for the parameter in the input field, click *Default value* (only available for some parameter types).
- To specify a possible default value, click *OK* without having selected or entered any value, or enter a '-' in the dialog's text input field.
- To cancel a command, click *Cancel* in any of the parameter dialogs, or enter an unallowed value in the dialog's text input field.

Selecting Signal Parameters

If a command parameter is an output signal and you select a signal which in turn have parameters, the signal parameters are asked for in a separate dialog. In this dialog, all parameters are listed with their default ("null") values:

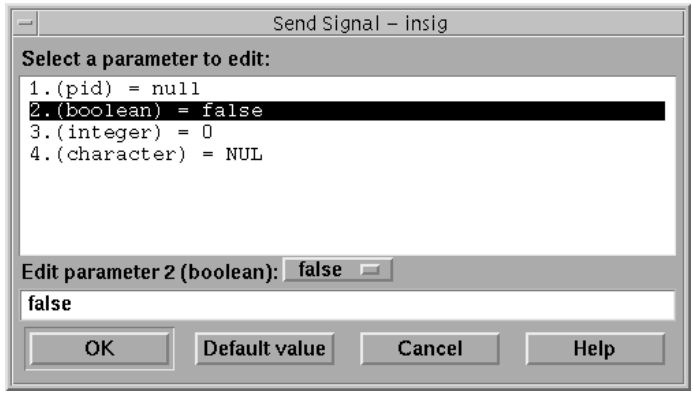


Figure 470: Editing signal parameters

To edit a signal parameter, you select the parameter and then edit the value displayed in the input field. If the parameter is of an enumerated type, the possible values can also be selected from the option menu just above the input field. By clicking *Default value*, the "null" value will be inserted in the input field. A changed parameter value is not updated in the parameter list until a parameter is selected in the list.

Click *OK* to accept all parameter values, and *Cancel* to abort the monitor command. The validity of the parameter values is not tested until you click the *OK* button.

Undo/Redo Commands

To undo a previous command, click the *Undo* quick button in the tool bar, or select *Undo* in the popup menu that is available in the text area. Then you get a dialog with all commands you have given so far. All commands are shown, even the ones entered using buttons or menu choices.

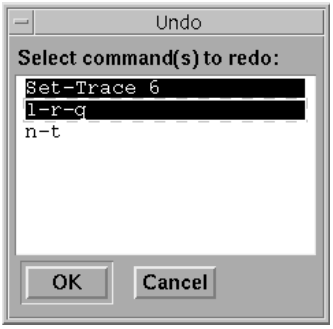


Figure 471: Undoing a command

Initially all commands except the last is selected. If you click *OK* the simulation is restarted and all the selected commands will be re-executed. By clicking on the commands in the dialog you can decide which commands you want to be re-executed in the restarted simulation.

Note:

The Undo function cannot handle to undo commands for setting or removing graphical breakpoints.

To redo a command in the current simulation, select *Command* in the popup menu in the text area. Then you also get a dialog with all commands you have given so far, in the same way as for *Undo*.

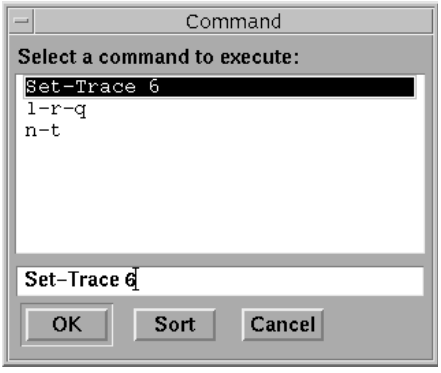


Figure 472: Redoing a command

After selecting one of the commands and clicking *OK* the command is executed. You can also change the command on the command line before clicking *OK*.

Customizing the Simulator UI

In the Simulator UI, you may change the contents and appearance of button modules and some additional windows. All these configurations are stored in a number of definition files, which are read at start-up of the Simulator UI. If you change any of the configurations, you are prompted to save them when you exit the Simulator UI.

Managing Command Buttons

You may wish to add your own command buttons for frequently used commands, or to change or delete existing command buttons. These operations are invoked from the *Group* menu in the button modules:

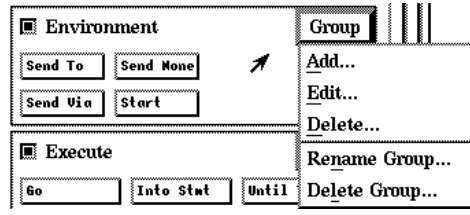


Figure 473: The Group menu

Adding a Button

To add a button to a module:

1. Select *Add* from the *Group* menu. A dialog is opened:

Figure 474: Adding a button

2. Enter a button label and a monitor command definition. The same syntax is used as when entering commands textually. If you omit a parameter value, it will be asked for in a parameter dialog when the command is executed.

3. Click *OK* to add the button and close the dialog. If you click *Apply*, the button is added and you can specify another button to add.

Note:

You cannot specify the location of the button in the module. It will be placed in the next available empty position.

Changing a Button

To change an existing button in a module:

1. Select *Edit* from the *Group* menu. A dialog is opened:

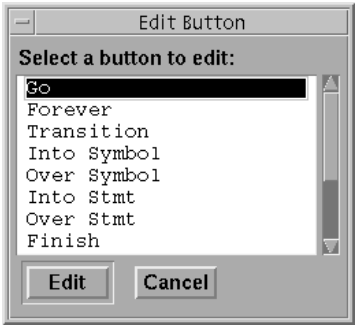


Figure 475: Selecting a button

2. Select the button to edit and click *OK*. A dialog similar to when adding a button is opened (see Figure 474).

3. Edit the button label and/or the command definition and click *OK*.

Deleting a Button

To delete an existing button in a module:

1. Select *Delete* from the *Group* menu. A dialog similar to when editing a button is opened (see Figure 475).
2. Select the button to delete and click *OK*. The button is deleted from the module and the remaining buttons are possibly re-arranged.

Managing Button Modules

In addition to the command buttons, you may also add, rename, delete and collapse/expand the button modules.

Collapsing and Expanding Modules

Modules may be collapsed and expanded to only show the command buttons of interest at the moment. A collapsed module hides all of its buttons and only displays the title bar:

Figure 476: A collapsed button module

To collapse and expand a module, click the toggle button to the left of the module name in the title bar.

To collapse and expand all modules, select *Collapse Groups* or *Expand Groups* in the *Buttons* menu.

Adding, Deleting and Renaming a Module

To add a new module to the bottom of the button module area, select *Add Group* in the *Buttons* menu. In the dialog, enter the module's name and click *OK*.

To delete a module, select *Delete Group* from the module's *Group* menu. In the confirmation dialog, click *OK*.

To rename a module, select *Rename Group* from the module's *Group* menu. In the dialog, enter a new name and click *OK*.

The Command and Watch Windows

In the Simulator UI, you can continuously view the internal status of the system by using the *Command* and *Watch* windows. These windows are opened from the *View* menu. They are both updated automatically whenever the monitor system becomes active.

The Command Window

In the Command window, you can execute a number of monitor commands automatically. The commands are executed in command modules, similar to button modules in the main window of the Simulator UI:

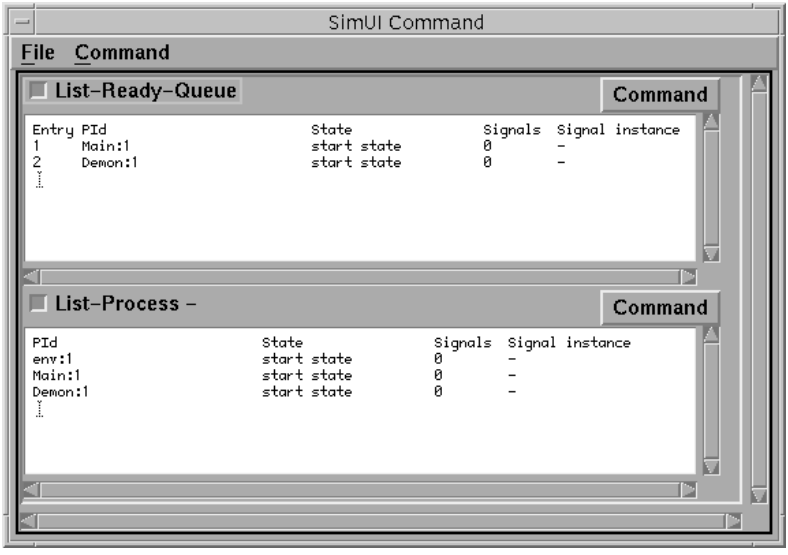


Figure 477: The Command window

By default, the commands List-Ready-Queue and List-Process are executed. You are advised to only use commands for examining the system, see "Examining the System" on page 2264.

The command modules are managed in the following ways:

- To add a command module, select *Command* from the *Command* menu in the menu bar. In the dialog, enter the monitor command to execute, including possible parameters.

- To change the command executed in a module, select *Edit* from the module's *Command* menu. In the dialog, enter a new command and click *OK*.
- To delete a command module, select *Delete* from the module's *Command* menu. No confirmation dialog is issued.
- To change the size of a module's text output area, select *Size* from the module's *Command* menu. In the dialog, use the slider to set the number of text lines to be visible.
- To collapse or expand a module, click the toggle button to the left of the module name in the title bar.

The Watch Window

In the Watch window, you can automatically monitor values of variables in the system. The variables are displayed on separate lines in the window:

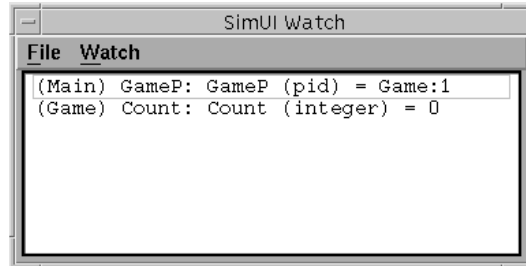


Figure 478: The Watch window

By default, the Watch window is empty. To add a variable, select *Add* from the *Watch* menu. In the dialog, enter the variable specification and click *OK*. A variable specification consists of the process instance name within parenthesis, followed by the variable name, e.g.
`(game:1) count`.

To change a variable specification, select *Edit* from the *Watch* menu. In the dialog, select the variable to edit and click *OK*. In the next dialog, you can edit the specification and click *OK*.

To delete a variable specification, select *Delete* from the *Watch* menu. In the dialog, select the variable to delete and click *OK*.

Tracing the Execution

When the SDL system is executing during simulation, you can obtain trace information of the execution. This makes it easier to follow the events that happen during execution.

There are three types of execution traces:

- *Textual Trace*, printed in the Simulator UI's text area, or on the terminal (in stand-alone mode)
- *GR Trace*, graphically illustrated by selecting SDL symbols in an SDL Editor
- *MSC Trace and Logging*, illustrated by drawing MSC events in an MSC Editor, or by saving the MSC events to a log file.

The amount of trace is determined by a trace value for each of the three trace types. The trace value controls the level of detail in the trace; the trace value 0 (zero) means no trace. Setting appropriate trace levels is often something you would like to do before the execution is started.

Apart from these continuous trace functions, there are also other possibilities to trace the execution. See "Other Tracing Functions" on page 2261.

Specifying Unit Names

Trace values can be assigned to different units of the SDL system; the system as a whole, blocks, process types, and process instances.

- When using the Simulator UI, you will be presented with a list of all units in the system, from which you may select the appropriate unit.
- When running the simulator stand-alone, you have to specify a unit, unless you want to set the trace for the system. In the trace commands described below, you may abbreviate the name of the unit. However, if several units match a unit name, the first unit found from the system level will be used. To make sure the correct unit is specified, the name can be prefixed with the diagram type or a qualifier.

Example 339: Specifying Trace Unit Names

```
System Demongame
   Block GameBlock
      Process Game
```

In this diagram structure, the unit name “Game” would refer to the block GameBlock. To specify the process Game instead, the unit could be expressed as “process Game” or “<<System Demongame / Block GameBlock>> Game”.

Determining the Scope of Trace

When a process instance executes a transition, or part of a transition, the trace value for the instance is found using the following algorithm:

1. If a trace value is defined for the process instance executing the transition, that value is used.
2. If not, and a trace value is defined for the process type, that value is used.
3. Otherwise, if a trace value is defined for the block enclosing the process, that value is used.
4. If still no trace value is found, the block structure is followed outwards until a unit is reached which has a trace value defined. The system always has a trace value defined.

For MSC trace the situation is slightly more complex; see “MSC Trace” on page 2259.

Textual Trace

Textual trace is printed for the transition or the SDL symbols that were last executed.

To quickly set the maximum textual trace level (6) for the system, click the *TEXT* button in the *Trace* module.

The general procedure to set the trace value for textual trace is:

1. Choose *Text Level* : *Set* in the *Trace* menu, or enter the command Set-Trace. The command takes two parameters, an optional unit name and the trace value.

2. Select or enter the name of the unit to set the trace for. If you do not specify a unit name, or the default name '-' is entered, the trace will be set for the whole system.

3. Select or enter a trace value between 0 and 6. The trace values are shortly described in the dialog, or after hitting `<Return>` in stand-alone mode. The following table explains them in more detail; for a full explanation, see *"Trace Limit Table" on page 2184 in chapter 49, The SDL Simulator*. The default textual trace value is 4 for the system.

Value	Trace Explanation and Examples
0	No trace
1	Trace of signals sent to the environment, as seen from the specified unit. This is useful if you only want to look at the external behavior of the unit. Example: `*  OUTPUT of Score from Game:1 to env:1` `*  Parameter(s) : -1`
2	Trace of transition start and timer outputs, i.e. what is causing the transition to occur. For transition starts, the process instance, initial state, input signal, sender process and value of Now is printed, e.g.: `*** TRANSITION START` `*  PId  : Demon:1` `*  State  : Generate` `*  Input  : T` `*  Sender : Demon:1` `*  Now  : 1.0000` For timer outputs, the timer name, receiver process and value of Now is printed, e.g.: `*** TIMER signal was sent` `*  Timer  : T` `*  Receiver : Demon:1` `*** Now  : 1.0000`

Value	Trace Explanation and Examples
3	As 2 + trace of important SDL actions, e.g. signal output, create, timer set and reset, nextstate and stop. Examples: `*  SET on timer T at 1.0000` `*  CREATE Game:1` `*  OUTPUT of Bump to Game:1` `*** NEXTSTATE  Generate`
4	As 3 + trace of other SDL actions as well, e.g. task, decision and export. Example: `*  ASSIGN  Count :=`
5	As 4 + result of actions, e.g. null transitions, implicit resets, and discarded signals.
6	As 5 + print of parameter values for signals, timers, create actions, etc. Examples: `*  ASSIGN  Count := 1` `*  OUTPUT of Score to env:1` `*  Parameter(s) : 1`

To list the textual trace values that are defined for the units in the system, choose *Trace Level : Show* in the *Trace* menu, or enter the command List-Trace-Values.

You can also reset the trace value for textual trace for a unit, i.e., set it to undefined. To do this, enter the command Reset-Trace and specify a unit (there is no button for this command).

GR Trace

GR trace will open an SDL Editor and continuously select the next SDL symbol to be executed in the corresponding process diagram. (See also "Tracing Simulations (Graphical Trace)" on page 1851 in chapter 43, *Using the SDL Editor*.)

To quickly set the GR trace level for the system to 1 (see below), click the *SDL* button in the *Trace* module.

The general procedure to set the trace value for GR trace is:

1. Choose *SDL Level : Set* in the *Trace* menu, or enter the command Set-GR-Trace. The command takes two parameters, an optional unit name and the trace value.
2. Select or enter the name of the unit to set the trace for. If you do not specify a unit name, or the default name '-' is entered, the trace will be set for the whole system.
3. Select or enter a trace value between 0 and 2. The trace values are shortly described in the dialog, or after hitting `<Return>` in stand-alone mode. The following table explains them in more detail. The default GR trace value is 0 for the system.

Value	Trace Explanation
0	No trace.
1	In an SDL Editor, show the next SDL symbol to be executed. This only happens when the monitor is activated the next time, i.e. when it becomes ready to accept a new command. No symbols are selected during execution when the monitor is inactive.
2	In an SDL Editor, follow the execution and show each SDL symbol as it is executed, until the monitor is activated again.

To list the GR trace values that are defined for the units in the system, choose *SDL Level : Show* in the *Trace* menu, or enter the command List-GR-Trace-Values.

You can also reset the trace value for GR trace for a unit, i.e., set it to undefined. To do this, enter the command Reset-GR-Trace and specify a unit (there is no button for this command).

MSC Trace and Logging

MSC trace will enable transformation of the SDL events that take place during execution into corresponding MSC events. Of course, not all SDL events are possible to transform into MSC events; typically, the events that can be transformed are sending and consumption of signals, setting and expiration of timers, and creation and termination of processes. By default, this transformation is enabled for the whole system.

The MSC events that are created by MSC trace can then be logged in two different ways:

- by opening an MSC Editor and continuously adding the events to an MSC diagram created for this purpose, or
- by continuously saving the events to a log file that later can be opened from an MSC Editor.

Therefore, setting the MSC trace value does not start the actual logging of MSC events in an MSC Editor or on file.

MSC Trace

To set the trace value for MSC trace:

1. Choose *MSC Level : Set* in the *Trace* menu, or enter the command Set-MSC-Trace. The command takes two parameters, an optional unit name and the trace value.
2. Select or enter the name of the unit to set the trace for. If you do not specify a unit name, or the default name '-' is entered, the trace will be set for the whole system.
3. Select or enter a trace value between 0 and 3. The trace values are shortly described in the dialog, or after hitting `<Return>` in stand-alone mode. The following table explains them in more detail; for a full explanation, see *"Message Sequence Chart Traces" on page 2187 in chapter 49, The SDL Simulator*. The default MSC trace value is 1 for the system.

Value	Trace Explanation
0	No trace.
1	For an MSC event that involves another unit, trace only if that unit's trace value is greater than 0.
2	For an MSC event that involves another unit, trace always, even if that unit's trace value is 0. Such an MSC event will be logged to interact with a special "void" instance in the MSC diagram.
3	Trace on block levels.

To list the MSC trace values that are defined for the units in the system, choose *MSC Level : Show* in the *Trace* menu, or enter the command List-MSC-Trace-Values.

You can also reset the trace value for MSC trace for a unit, i.e., set it to undefined. To do this, enter the command Reset-MSC-Trace and specify a unit (there is no button for this command).

Logging of MSC Events

To start the continuous logging of MSC events in an MSC Editor:

1. Choose *MSC Trace : Start* in the *Trace* menu, or enter the command Start-Interactive-MSC-Log. The command takes one parameter, a symbol level determining the amount of information that should be part of the MSC.
2. Select or enter a symbol level between 0 and 2 (see the table below).

   An MSC Editor is opened and the execution is then traced by adding MSC events to the created diagram. (See also "Tracing a Simulation in a Message Sequence Chart" on page 1725 in chapter 39, *Using Diagram Editors*.)

To quickly start the continuous logging of MSC events in an MSC Editor with symbol level 1, click the *MSC* button in the *Trace* module.

To start the continuous logging of MSC events in a log file:

1. Choose *MSC Trace : Start Batch* in the *Trace* menu, or enter the command Start-Batch-MSC-Log. The command takes two parameters, a symbol level and the name of the log file. The symbol level determines the amount of information that should be logged.
2. Select or enter a symbol level between 0 and 2 (see the table below).
3. Select or enter a log file name, preferably with the suffix `.mpr`.

When an MSC log is started, the amount of information that should be part of the log can be decided by giving the symbol level parameter an appropriate value, according to the table below:

Level	Information in MSC
0	Basic MSC, i.e. containing events for signals and timers plus create and stop.
1	Basic MSC extended with condition symbols for each next-state.
2	Basic MSC extended with condition symbols for each next-state and action symbols for task, decision, call, and return.

Note that you cannot have both types off logging enabled at the same time. To see what type of logging is enabled, enter the command List-MSC-Log (this command has no button).

To stop the logging, choose *MSC Trace : Stop* in the *Trace* menu, or enter the command Stop-MSC-Log. If you used logging to an MSC Editor, you should then save the trace from the editor. If you used logging to file, the file is automatically saved and closed.

Other Tracing Functions

Showing SDL Symbols

If you do not have GR trace enabled, it is still possible to show the next symbol to be executed in an SDL Editor. To do this, choose *Next Symbol* in the *Show* menu, or enter the command Show-Next-Symbol.

In a similar fashion, the last executed symbol can be shown in an SDL Editor. This will give you a correspondence between textual trace and the SDL Editor, since textual trace always is printed for the last executed symbols. To do this, choose *Prev Symbol* in the *Show* menu, or enter the command Show-Previous-Symbol.

Tracing Back to C Code

A trace-back to the generated C source code for the system is also possible. To show where in the C code the execution is at the moment, choose *C Line* in the *Show* menu, or enter the command Show-C-Line-Number. The file name and line number of the C source file is printed, and the C source file is opened in the Text Editor, positioned on the correct line.

Generating Coverage Information

To see how much of the SDL system you have covered during simulation, you can open the coverage information in a Coverage Viewer, and for instance examine which parts of the system that have not been executed during the simulation. To start the Coverage Viewer, choose *Coverage* in the *Show* menu, or enter the command Show-Coverage-Viewer.

Executing a Simulator

There are a number of monitor commands for executing one or more transitions in process instances, and for stepping symbol by symbol within a transition.

> **Note:**
>
> During execution, you may be prompted for input before the execution can continue. See *"Run-time Prompting" on page 2197 in chapter 49, The SDL Simulator* for more information.

Continuous Execution

To start executing the simulation program continuously, click the *Go* button in the *Execute* module, or enter the command Go. The execution continues until one of the conditions listed in "Activating the Monitor" on page 2241 becomes true, for instance when reaching a breakpoint. See also "Stopping the Execution" on page 2264.

> **Note:**
>
> The button *Forever*, corresponding to the command Go-Forever, behaves very similar to the above. The difference is that the monitor **does not** become active when the system is completely idle. This feature is valuable when communicating with other simulations or applications.

Executing Until a Condition

To execute until a certain point in time:

1. Choose *Until Time* in the *Execute* menu, or enter the command Proceed-Until. The command takes one parameter, the value of the simulation time when to stop executing.

2. Enter a time value, either an absolute value (without sign) or a value relative to Now (with a '+' sign). That is, "7.5" is an absolute time value, whereas "+7.5" is the time value Now+7.5.

To execute until, but not including, the next timer output, click the *Until Tim*er button in the *Execute* module, or enter the command Proceed-To-Timer.

Executing Transitions

To execute the next transition, or the remainder of the current transition, click the *Transition* button in the *Execute* module, or enter the command Next-Transition.

To execute a number of transitions until a transition with a textual trace value > 0 has been executed, choose *Until Trace* in the *Execute* menu, or enter the command Next-Visible-Transition. This is very valuable when you have set the trace value for some "uninteresting" parts of the system to 0, and you want to skip over those parts.

Single-Stepping Symbols or Statements

To execute only the next SDL symbol, choose *Over Symbol* in the *Execute* menu, or enter the command Next-Symbol. This may execute several statements, and will step over procedure calls.

To execute only the next SDL statement, choose *Over Stmt* in the *Execute* menu, or enter the command Next-Statement. This will step over procedure calls.

Executing Procedures

The only way to follow the execution into a procedure is to use the following single-stepping functions.

To execute only the next SDL symbol, and to step into possible procedure calls, click the *Symbol* button in the *Execute* module, or enter the command Step-Symbol. This may execute several statements.

To execute only the next SDL statement, and to step into possible procedure calls, choose *Into Stmt* in the *Execute* menu, or enter the command Step-Statement.

To execute a procedure up to and including its return, choose *Finish* in the *Execute* menu, or enter the command Finish. In a process, this command behaves exactly like Next-Transition.

Stopping the Execution

To stop the execution of transitions and symbols manually:

- In the Simulator UI, click the *Break* button in the *Execute* module.
- In stand-alone mode, press `<Return>` during printing of trace information (repeatedly, if necessary). No other characters may be typed before `<Return>` is pressed.

Caution!

There are some situations in which it is not possible to stop the execution in this way, for instance in an endless loop within an SDL symbol. The only way to stop the simulator in these situations is to terminate the simulation program from the operating system, for instance by pressing `<Ctrl+C>`, in which case all simulation results are lost.

Examining the System

In the Simulator, you can issue a number of monitor commands to examine the internal status of the system. Detailed information concerning processes, procedure calls, signals, timers, and variables can be requested. Using the Simulator UI, you may continuously view the internal status by using the Command and Watch windows; see "The Command and Watch Windows" on page 2252.

Current Process and Scope

Some of the commands used for examining the system operate on a specific process instance, the *current process*, identified by the current *scope*. A scope is a reference to a process instance, a reference to a service instance if the process contains services, and possibly a reference to a procedure instance called from this process/service (the *current procedure*). The scope is also used by some of the commands for modifying the system; see "Modifying the System" on page 2280.

The scope is automatically set by the execution commands, when entering the monitor, to the next process instance in turn to execute. You may change the scope if you would like to examine (or modify) another process, service or procedure instance. Changing the scope does not change the execution order of process instances. As soon as the execution continues again, the scope is reset to the next process instance in turn to execute.

The Process/service Scope

To print the current process/service scope enter the command Scope.

To set the current process/service scope:

1. Choose *Set Scope* in the *Examine* menu, or enter the command Set-Scope. This command takes one parameter, a process instance, and optionally if the process contains services, a second parameter which specifies a service name.
2. Select or enter the name of a process instance.
3. If the process instance contains services, select or enter the name of a service instance.

The scope is set to the specified process/service, at the bottom procedure call.

The Procedure Scope

To print the procedure call stack for the process/service instance defined by the current scope, choose *Call Stack* in the *Examine* menu, or enter the command Stack.

To change the procedure scope within the current process/service scope, you can move the scope one step up or down in the procedure call stack by entering the command Up or Define-MSC-Trace-Channels. Going up from a service leads to the process containing the service. To go down in a service within a process, select or enter the name of the service instance.

Printing the Simulation Time

To print the current value of the simulation time, in case it is not displayed by the textual trace, choose *Now* in the *Examine* menu, or enter the command Now.

Printing the Process Ready Queue

The ready queue is the queue of process instances that are ready to execute a transition, i.e., those instances that have received a signal that can cause an immediate transition, but that have not yet had the opportunity to execute this transition to its end.

To print an ordered list of the process instances in the ready queue, choose *Ready Q* in the *Examine* menu, or enter the command List-Ready-Queue. The list contains an entry number (queue position), the process instance, its current state, the number of signals in its input port, and the name of the signal that will cause the next transition. If a process instance has active procedure calls, the current executing procedure instance is also listed. If the state name is followed by a '*', then the process/procedure is currently executing a transition starting from this state.

The ready queue is by default printed in the Command window. It may look like this:

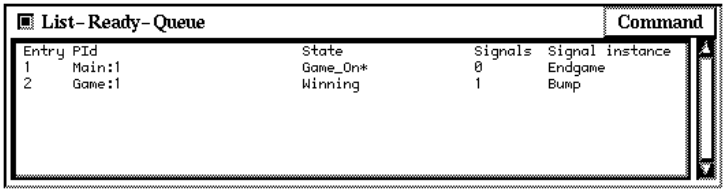


Figure 479: The ready queue in the Command window

Examining Process Instances

To list all active process instances of a certain process type:

1. Click the *Process List* button in the *Examine* module, or enter the command List-Process. The command takes one parameter, the name of the process type. (The button already has the parameter '-'.)

2. Select or enter a process type. If no process type is specified, or it is specified as '-', all active process instances in the system are listed. The list contains the same details as described for the process ready queue, above.

To print information about the current process instance, enter the command Examine-PId. The information contains the current values of Par-

ent, Offspring, Sender and a list of all currently active procedure calls made by the process instance (the stack). For more information on current process and procedure call stack, see "Current Process and Scope" on page 2264. An example of output is:

```
Parent    : null
Offspring : Game:1
Sender    : env:1

PId       Main:1
```

Examining Signal Instances

To list all signal instances in the input port of the current process instance, choose *Input Port* in the *Examine* menu, or enter the command List-Input-Port. The list contains an entry number (queue position), the signal type, and the sending process instance. If the entry number is prefixed by a '*', the signal instance is the one to be consumed in the next transition performed by the process instance. For more information on the current process, see "Current Process and Scope" on page 2264. An example of output is:

```
Input port of Game:1
Entry   Signal name          Sender
*1      Bump                 Demon:1
 2      GameOver             Main:1
```

To print the parameters of a signal instance in the input port of the current process instance:

1. Choose *Signal* in the *Examine* menu, or enter the command Examine-Signal-Instance. This command takes one parameter, an entry number in the input port.
2. Enter an entry number. To see what entry number is associated with the signal instance, list the input port as described above. An example of output is:

```
Signal name : Score
Parameter(s) : -1
```

Examining Timer Instances

To list all active timers in the system, choose *Timer List* in the *Examine* menu, or enter the command List-Timer. The list contains an entry number, the timer name, the corresponding process instance, and the associated time. An example of output is:

```
Entry   Timer name        PId                Time
1       T                 Demon:1            4.0000
```

To print the parameters of an active timer instance:

1. Enter the command Examine-Timer-Instance. This command takes one parameter, an entry number in the timer list.
2. Enter an entry number. To see what entry number is associated with the timer instance, list the active timers as described above. If only one timer is active, the entry number is not needed.

Examining Variables

To print the value of a variable or formal parameter in the current process or procedure:

1. Click the *Var In Scope* button in the *Examine* module, or enter the command Examine-Variable. This command takes two optional parameters, the name of the variable and a specification of a variable component.
2. Select or enter the name of a variable, possibly abbreviated. If no variable name is specified, all variables and formal parameters of the current process or procedure are printed.

   If a variable has several components (e.g. an array, struct or string) and it is specified without a component, the values of all components of the variable is printed (e.g. the complete array).

   – To print only a certain component, specify the component after the variable name. (In the Simulator UI, this must be done on the text input line in the dialog, after a variable name has been selected.)

   – To get a list of the possible components of a variable, enter a '?' after the variable name. (In the Simulator UI, this must be done on the text input line in the dialog, after a variable name has

been selected; another dialog is then opened, in which the component can be selected.)

More information on the input and output of SDL data types can be found in *"Input and Output of Data Types" on page 2137 in chapter 49, The SDL Simulator*.

To examine variables or formal parameters in any process, click the *Variable* button in the *Examine* module. In addition to the variable name and component (as described above), you will be prompted for the process name.

Managing Breakpoints

By setting breakpoints in the system, the execution of a simulator can be stopped and the monitor system activated at a certain point of interest. They can be used to trap certain SDL symbols, transitions, signal outputs, and variable changes.

Breakpoints are often used together with continuous execution to reach a certain state of the system (see "Continuous Execution" on page 2262). The monitor system becomes active when the breakpoint condition becomes true. Care should be taken when using continuous execution with breakpoints; if the breakpoint is never reached, the system may continue executing "forever."

Breakpoint Commands

When defining a breakpoint, you may specify one or more monitor commands to be executed after the breakpoint has been reached. In this way it is possible to, for instance, automatically print information about the system by using the examine commands. The monitor commands are specified using the same syntax as when entering them textually.

If you want to use several monitor commands, they have to be separated by spaces and semicolons. It is even possible to use the execution commands in this context to automatically continue the execution after a breakpoint has been reached. However, care should be taken to make sure the system does not end up executing "forever." An example of a combination of breakpoint commands may be:

```
Examine-PId ; Go
```

Setting a Symbol Breakpoint

Symbol breakpoints are set on a specific SDL symbol. Symbol breakpoints are checked **before** a symbol is executed, i.e., the symbol is not executed when the breakpoint is reached.

There are two ways to set a symbol breakpoint, either graphically by selecting the SDL symbol directly in the SDL Editor, or textually by specifying an SDT reference to the SDL symbol.

When a symbol breakpoint has been set, the breakpoint is indicated by a red “stop” sign placed at the SDL symbol in the SDL diagram.

To set a symbol breakpoint graphically using the SDL Editor:

1. If necessary, open an SDL Editor on the diagram containing the SDL symbol.
2. Choose *Connect sdle* in the *Breakpoint* menu in the Simulator UI, or enter the command Connect-To-Editor. A new menu *Breakpoints* now appear in the SDL Editor.
3. Select the symbol in the SDL Editor.
4. From the *Breakpoints* menu in the SDL Editor, choose one of the two *Set Breakpoint* commands. The first command opens a dialog to allow entering one or more optional breakpoint commands. The second command sets the breakpoint without any breakpoint command.

To set a symbol breakpoint textually using an SDT reference:

1. Choose *Symbol* in the *Breakpoint* menu, or enter the command Breakpoint-At. This command takes two parameters, a textual SDT reference to the SDL symbol and an optional breakpoint command.

Setting a Transition Breakpoint

Transition breakpoints are set on a transition, i.e. the combination of a process instance, state, signal and sender. Transition breakpoints are checked **before** a transition is executed, i.e., the transition is not executed when the breakpoint is reached.

To set a transition breakpoint:

1. Choose *Transition* in the *Breakpoint* menu, or enter the command Breakpoint-Transition. This command takes the following parameters: the process name and instance number, the name of a service, the state of the process/service, the signal, the sender's process name and instance number, a breakpoint counter, and an optional breakpoint command.

   Any of the parameters may be omitted or given the value '-', which means that any value will match that parameter. In this way, transition breakpoints can be matched by more than one transition.

2. Select or enter the name of the process in which the transition exists. If no process is specified, any process will match.

3. Enter the instance number of the process. If no instance number is specified, any instance of the process specified above will match.

4. Select or enter the name of the service of interest.

5. Select or enter the state of the process where the transition is executed from. If no state is specified, any state will match.

6. Select or enter the name of the signal that causes the transition. If no signal is specified, any signal will match.

7. Select or enter the name of the process sending the signal. If no process is specified, any sender process will match. Note that "env" may be entered for the environment process.

8. Enter the instance number of the sender process. If no instance number is specified, any instance of the process specified above will match.

9. Enter a breakpoint counter, i.e. how many times the breakpoint condition must be true before the execution is stopped. If no counter value is specified, the default value 1 is used.

10. Enter one or more optional breakpoint commands.

Setting an Output Breakpoint

Output breakpoints are set on a signal output, i.e. the combination of a signal, sender process and receiver process. Output breakpoints are checked immediately **after** a signal is sent.

To set an output breakpoint:

1. Choose *Output* in the *Breakpoint* menu, or enter the command Breakpoint-Output. This command takes the following parameters: the signal name, the sender's process name and instance number, the receiver's process name and instance number, a breakpoint counter, and an optional breakpoint command.

   Any of the parameters may be omitted or given the value '-', which means that any value will match that parameter. In this way, output breakpoints can be matched by more than one signal output.

2. Select or enter the name of the signal. If no signal is specified, any signal will match.

3. Select or enter the name of the process sending the signal. If no process is specified, any sender process will match. Note that "env" may be entered for the environment process.

4. Enter the instance number of the sender process. If no instance number is specified, any instance of the process specified above will match.

5. Select or enter the name of the process receiving the signal. If no process is specified, any receiver process will match. Note that "env" may be entered for the environment process.

6. Enter the instance number of the receiver process. If no instance number is specified, any instance of the process specified above will match.

7. Enter a breakpoint counter, i.e. how many times the breakpoint condition must be true before the execution is stopped. If no counter value is specified, the default value 1 is used.

8. Enter one or more optional breakpoint commands.

Setting a Variable Breakpoint

Variable breakpoints are set on a specific variable and are triggered whenever the value of the variable is changed. Variable breakpoints are checked **after** a variable assignment, i.e., the execution stops immediately after the symbol or assignment statement where the value was changed.

To set a breakpoint on a variable in the current process:

1. Choose *Variable* in the *Breakpoint* menu, or enter the command Breakpoint-Variable. This command takes two parameters: the variable name and an optional breakpoint command.
2. Select or enter a variable name in the current process.
3. Enter one or more optional breakpoint commands.

Listing and Removing Breakpoints

To list all defined breakpoints, choose *List* in the *Breakpoint* menu, or enter the command List-Breakpoints. The list contains an entry number and the break condition for each breakpoint, i.e. the values of the parameters specified when the breakpoint was defined. An omitted parameter value, where permitted, is listed as “any.”

A symbol breakpoint can be visualized by opening the SDL diagram and selecting the SDL symbol in an SDL Editor. To see where a specific symbol breakpoint has been defined:

1. Enter the command Show-Breakpoint (this command has no associated button or menu choice). This command takes one parameter, an entry number in the breakpoint list.
2. Enter an entry number. (To see what entry number is associated with the symbol breakpoint, list the breakpoints as described above.) The SDL symbol, pointed out by the SDT reference specified for the breakpoint, becomes selected in an SDL Editor.

To remove a defined breakpoint using the Simulator UI:

1. Choose *Remove* in the *Breakpoint* menu, or enter the command Remove-All-Breakpoints. This command takes one parameter, an entry number in the breakpoint list.
2. Enter an entry number. (To see what entry number is associated with the symbol breakpoint, list the breakpoints as described above.) The breakpoint is removed.

To remove all symbol breakpoints set on an SDL symbol:

1. If necessary, open an SDL Editor on the diagram containing the SDL symbol.
2. If the *Breakpoints* menu is not visible in the SDL Editor menu bar, choose *Connect sdle* in the *Breakpoint* menu in the Simulator UI, or enter the command Connect-To-Editor. The menu *Breakpoints* now appear in the SDL Editor.
3. Select a symbol with a red breakpoint symbol in the SDL Editor.
4. From the *Breakpoints* menu in the SDL Editor, choose the *Remove Breakpoint* command. All symbol breakpoints are removed.

To remove all defined breakpoints:

- Enter the command Remove-All-Breakpoints (this command has no associated button or menu choice). All defined breakpoints are now removed.

Sending Signals from the Environment

Usually, you have to send signals into the system to make something of interest happen, for instance when the system is completely idle. You can send signals directly to a process instance, or indirectly by specifying a channel.

Sending Signals to a Process

To send a signal from the environment to a certain process:

1. Click the *Send To* button in the *Send Signal* module, or enter the command Output-To. This command takes three parameters: the signal name, any signal parameters, and the process instance of the receiver.
2. Select or enter the name of the signal to send.
3. If the signal has parameters, set the values of the parameters in the dialog that is opened. Select a parameter in the list, enter/edit its value using the input line, or the option menu containing possible values (for enumeration type parameters). Click the *Default value* button to get a “null” value for the parameter.
4. Select or enter the name of a process instance to send the signal to.

   The Simulator tries to find a path of channels and signal routes to the receiver. If no path was found, an error message is printed. If a path was found, the signal is either successfully sent (and placed in the input port of the receiver), or is discarded (an immediate null transition occurred), i.e. the receiver is in a state that cannot receive or save the specified signal.

Sending Signals via a Channel

To send a signal from the environment via a channel:

1. Click the *Send Via* button in the *Send Signal* module, or enter the command Output-Via. This command takes three parameters: the signal name, any signal parameters, and an optional channel.
2. Select or enter the name of the signal to send.
3. If the signal has parameters, set the values of the parameters in the dialog that is opened. Select a parameter in the list, enter/edit its val-

ue using the input line, or the option menu containing possible values (for enumeration type parameters). Click the *Default value* button to get a "null" value for the parameter.

4. Select or enter the name of a channel to send the signal via. If no channel is specified, any channel from the environment may be used.

   The Simulator tries to find a single process instance that can receive the signal. If no or several possible receivers were found, an error message is printed. If a single receiver was found, the signal is either successfully sent (and placed in the input port of the receiver), or is discarded (an immediate null transition occurred), i.e. the receiver is in a state that cannot receive or save the specified signal.

Causing a Spontaneous Transition

To send a "none" signal to a process, i.e. to try to cause a spontaneous transition in the process:

1. Click the *Send None* button in the *Send Signal* module, or enter the command Output-None. This command takes one parameter: the process name. If the process contains services, it is also necessary to specify one of the services.

2. Select or enter the name of the process to send a "none" signal to. A message is printed, indicating whether a spontaneous transition occurred.

Logging the Execution

The following types of interaction during simulation can be logged to file:

- The commands issued by the user.
- The complete interaction with the user, i.e. commands issued and resulting print-outs. In other words, all texts visible in the Simulator UI's text area or in the terminal window.
- The signals sent to, from and via a unit in the system.

> **Note:**
> The logging of MSC events to file is described in "MSC Trace and Logging" on page 2258.

Logging Commands

To start logging of all issued monitor commands from now on:

1. Select *Start Command Log* from the *Log* menu, or enter the command Command-Log-On. This command takes one parameter, the name of the log file.
2. Select or enter a log file name; **on UNIX** preferably with the suffix `.com`.

To stop the logging, select *Stop Command Log* from the *Log* menu, or enter the command Command-Log-Off.

After logging has been stopped, it can be continued on the same log file, i.e. the existing log file is appended, or on a new log file.

To continue logging of monitor commands:

- If using the Simulator UI's menu commands, select *Start Command Log* from the *Log* menu. Select or enter the same file name as before, or a new file name.
  - If a new file name was specified, the logging is continued on the new file.
  - If the name of an existing log file was specified, a dialog is opened:

Figure 480: Continued logging on an existing file

Click *Append* to append the existing log file. Click *Create* to overwrite the existing log file.

- If using monitor commands, enter the command Command-Log-On, with or without a file name parameter.
  - If you do not specify any file name, the logging is continued on the same lag file as before.
  - If a file name is specified, the logging is continued on this file.

All monitor commands issued by the user so far in the Simulator UI session can also be saved on a command file, without starting continuos logging of commands. To do this, select *Save* from the popup menu on the input line and select a command file.

A command log file may later be read in and executed to repeat the same command session. To do this:

1. Choose *Script* in the *Execute* menu, or enter the command Include-File. This command takes one parameter: the name of a file containing monitor commands.
2. Select or enter the name of an existing command file. When the file has been specified, it is read and the commands in the file are executed exactly as they are stated in the file.

Logging the User Interaction

Complete logging of all user interaction works in the same way as logging of monitor commands, described above. The only difference is the name of the menu choices and commands. Either select *Start Complete Log* and *Stop Complete Log* from the *Log* menu, or enter the commands Log-On and Log-Off.

Logging Signal Interaction

The signals that are sent to, from or via a unit in the system can be logged on file. The signal log file will contain a line for each signal sent, including the simulation time, the name of the signal, the sender process instance and the receiver process instance. Values of signal parameters are also printed.

More than one signal log can be active at a time, each one in a separate log file.

To start a signal log:

1. Enter the command Signal-Log (this command has no associated button). This command takes two parameters: a unit in the system, and the name of a log file.
2. Select or enter a unit whose signals are to be logged. A unit can be one of the following:
   - A channel or signalroute. Signals sent through the channel or signalroute are logged.
   - A process type or process instance. Signals sent to and from any of the process instances are logged.
   - A block or a system. Signals sent within, to and from the system or block are logged.
   - The process “env.” In this way, the signal interface between the system and the environment is logged.
3. Select or enter a file name of the signal log file.

To list the active signal logs, enter the command List-Signal-Log (this command has no associated button). The list contains an entry number, the unit name and the log file name.

To stop a signal log:

1. Enter the command Close-Signal-Log (this command has no associated button). This command takes one parameter, an entry number in the signal log list.
2. Enter an entry number. To see what entry number is associated with the timer instance, list the active signal logs as described above. If only one signal log is active, the entry number is not needed.

Modifying the System

The are a number of commands that change the behavior of the simulated system. These commands are useful to make debugging easier. They can, for example, be used to recover from logical errors, or to put the system in a state that could be difficult to achieve otherwise.

Caution!

When these commands are used, **it is no longer the original system that is simulated**, as the commands modify the behavior of the system. It is completely up to the user to interpret the simulation and its relation to the original system.

Some of these commands operate on the current process. For more information on the current process, see "Current Process and Scope" on page 2264.

Sending an Internal Signal

To simulate the sending of an internal signal between two process instances:

1. Enter the command Output-Internal. This command takes the following parameters: the name of the signal, values of any signal parameters, receiver process instance, and sender process instance.
2. Select or enter the name of the signal to send.
3. If the signal has parameters, set the values of the parameters in the dialog that is opened. Select a parameter in the list, enter/edit its value using the input line, or the option menu containing possible values (for enumeration type parameters). Click the *Default value* button to get a "null" value for the parameter.
4. Select or enter the receiver process instance. Note that "env" may be entered for the environment process.
5. Select or enter the sender process instance. Note that "env" may be entered for the environment process.

The signal is put in the input port of the receiver process, if it is in the valid input signal set; otherwise, it is discarded. No checks are made whether a path exists between the sender and the receiver.

Changing the Process State

To change the state of the current process, service, or procedure:

1. Choose *State* in the *Change* menu, or enter the command Nextstate. This command takes one parameter: the name of the new state.
2. Select or enter the name of the state to go to.

Creating and Stopping Processes

To create a new process instance:

1. Choose *Create Process* in the *Change* menu, or enter the command Create. This command takes the following parameters: the process to create, the parent process instance, and any process parameters.
2. Select or enter the name of the process to create a new instance for.
3. Select or enter the parent process instance. “env” may be entered for the environment process, and “null” may also be entered.
4. If the process has parameters, enter the values of the parameters one by one.

   If the number of instances already is greater than or equal to the maximum number for this process type, you have to verify the create action. The following message appears:

   ```
   Attempt to create more than the max number of
   concurrent instances.
   Do you still want to create the instance :
   ```

5. Enter ‘y’ or ‘Y’ to create; all other answers cancels the create.

To stop a running process instance:

1. Choose *Stop Process* in the *Change* menu, or enter the command Stop. This command takes one parameter, the process instance to stop.
2. Select or enter the name of the process instance to stop.

Setting and Resetting Timers

To set a timer in the current process:

1. Choose *Set Timer* in the *Change* menu, or enter the command Set-Timer. This command takes the following parameters: the name of the timer, any timer parameters, and a time value.
2. Select or enter the name of a timer in the current process.
3. If the timer has parameters, enter the values of the parameters one by one.
4. Enter a time value when the timer is to expire, either an absolute value (without sign) or a value relative to Now (with a '+' sign). That is, "7.5" is an absolute time value, whereas "+7.5" is the time value Now+7.5.

To reset a timer in the current process:

1. Choose *Reset Timer* in the *Change* menu, or enter the command Reset-Timer. This command takes the following parameters: the name of the timer, and any timer parameters.
2. Select or enter the name of a timer in the current process.
3. If the timer has parameters, enter the values of the parameters one by one.

Changing a Variable

To change the value of a variable in the current process:

1. Choose *Variable* in the *Change* menu, or enter the command Assign-Value. This command takes the following parameters, the name of the variable, an optional specification of a variable component, and the new value.
2. Select or enter the name of a variable, possibly abbreviated.

   If a variable has several components (e.g. an array, struct or string) and it is specified without a component, the values of all components of the variable have to be specified (e.g. the complete array).

To assign a component only, the same rules as for examining a variable apply (see "Examining Variables" on page 2268):

- To assign only a certain component, specify the component after the variable name. (In the Simulator UI, this must be done on the text input line in the dialog, after a variable name has been selected.)
- To get a list of the possible components of a variable, enter a '?' after the variable name. (In the Simulator UI, this must be done on the text input line in the dialog, after a variable name has been selected; another dialog is then opened, in which the component can be selected.)

More information on the input and output of SDL data types can be found in *"Input and Output of Data Types" on page 2137 in chapter 49, The SDL Simulator*.

Changing the Input Port

To remove a signal instance in the input port of the current process:

1. Enter the command Remove-Signal-Instance. This command takes one parameter, an entry number in the input port.
2. Enter an entry number. To see what entry number is associated with the signal instance, list the input port as described in "Examining Signal Instances" on page 2267.

> **Note:**
>
> Entry numbers are just positions in the input port. The removal of a signal changes the entry numbers of the remaining signals.

To change the placing of a signal instance in the input port of the current process:

1. Choose *Input Port* in the *Change* menu, or enter the command Rearrange-Input-Port. This command takes two parameters, the current and new entry number in the input port.
2. Enter the current entry number of the signal instance. To see what entry number is associated with the signal instance, list the input port as described in "Examining Signal Instances" on page 2267.
3. Enter the new entry number of the signal instance. The current signal instance at this position will be moved down in the queue.

Rearranging the Ready Queue

You can change the order of process instances in the ready queue, i.e. the execution order of processes.

To change the placing of a process instance in the ready queue:

1. Choose *Ready Q* in the *Change* menu, or enter the command Rearrange-Ready-Queue. This command takes two parameters, the current and new entry number in the ready queue.
2. Enter the current entry number of the process instance. To see what entry number is associated with the process instance, list the ready queue as described in "Printing the Process Ready Queue" on page 2266.
3. Enter the new entry number of the process instance. The current process instance at this position will be moved down in the queue.

Exiting a Simulator

To exit a simulator, choose *Stop Sim* in the *Execute* menu, or enter the command Exit or Quit. If the command is abbreviated, you have to confirm the exit operation; the following message appears:

```
Do you really want to exit program :
```

Enter 'y' or 'Y' to exit; all other answers cancels the exit.

To exit the Simulator UI, select *Exit* from the *File* menu. If you have changed the configuration of the Simulator UI, you are asked whether to save these changes. See "Customizing the Simulator UI" on page 2249. If you save them under the default file names, they are read in and restored the next time the Simulator UI is started.

Chapter

51 *Remote Target Simulation*

This chapter describes how to run a simulator on a target and connect it to the SDL simulator interface (SimUI) on the host via TCP/IP communication.

Target Simulation

Overview

A simulator executable (called *simulator*) communicates with the SDL Suite tools, e.g. the SDL Simulator UI (SimUI), via the SDL Suite communication mechanism (the PostMaster). By using some special features it is possible to run the simulator on a target system (called *target*), while the SDL Suite tools and Postmaster run on a host system (called *host*) separate from the target. This set-up is known as *target simulation*. This gives the possibility to test a simulator in its correct environment and still get the trace and debug possibilities offered by the SDL Suite.

For instance, the target could be a VME147 bus system running VxWorks, and the host is a Sun workstation running SunOS 5. The simulator running on the target is built using a special library, instead of the ordinary simulator library containing the simulator kernel. The special library contains the kernel for a simulator on VxWorks, as well as a communication interface to the host. On the host, the simulator is replaced by a gateway executable (called *gateway*) that communicates with the simulator on the target over the TCP sockets connection. Figure 481 shows the set-up for target simulation.

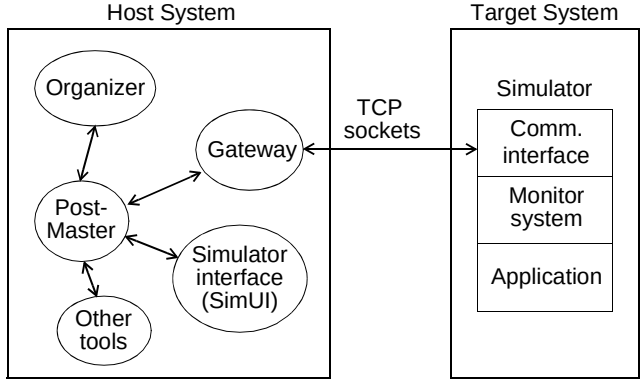


Figure 481: Target simulation set-up

Architecture

Simulators, as well as other executables communicating via the Postmaster, has a set of layers as shown in Figure 482, where the layers are represented by the files that implement them. All units (the Simulator, Postmaster and SimUI) are located on the same host machine.

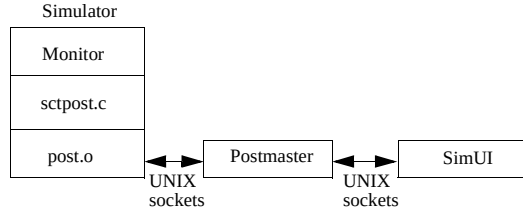


Figure 482: Architecture of an ordinary simulator

`sctpost.c`, `post.o` (`post64.o` for solaris 64-bit) and a few other files are then linked into the simulator kernel used for making the simulator.

The basic principle of this target simulation is to direct the communication between the simulator and the Postmaster over a TCP sockets connection. This re-direction must be transparent to the simulator and to the Postmaster.

To direct the messages between the Simulator Monitor and the Postmaster through the TCP sockets, a new layer, implemented in the file `tlayer.c`, is introduced between `sctpost.c` and `post.o` according to Figure 483.

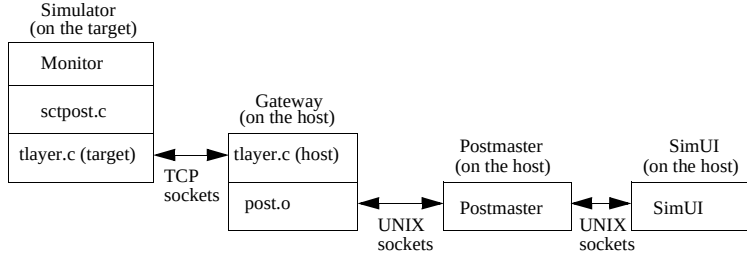


Figure 483: tlayer.c is introduced

For the target, `tlayer.c` emulates `post.o` (the same interface), and for the host, `tlayer.c` emulates `sctpost.c` (the same interface). (**In Windows NT** there are other communication mechanisms on host than UNIX sockets.)

The Configuration File and Simulator UI Command Extension

All configuration is made on host side in a file called `tarsim.cfg`. This file is searched by `host_smr.sct/host_smr.exe` in a directory specified by the environment variables (searched in this order):
Unix: `$TARSIMPATH, $PWD, $HOME, $sdtdir`.
Windows NT: the only searched variables are `%TARSIMPATH%`. You set the environment variable `%TARSIMPATH%` as described below.

1. Open the System Control Panel and select the *Environment* tab.
2. In the *Variable* field enter **`TARSIMPATH`**, and in the *Value* field enter the directory (including logical disk drive) where `tarsim.cfg` is located.
3. Press *Set* and close down the System Control Panel.

The syntax of `tarsim.cfg` is described below. In the delivery, a sample `tarsim.cfg` is supplied.

- All lines beginning with '#' (hash) are considered as comments and left unread.
- All lines after the optional command END are left unread, regardless of contents.
- All commands must be at the first position in a line.
- All commands and arguments are case insensitive, except arguments specifying files, etc.
- Separation between commands and arguments are any whitespace (i.e. space or tab).

Example 340: A sample `tarsim.cfg` ----------

```
# targsim.cfg, config file for simulating an SDT executable simulator
# on remote target with SDT SIMUI on a host.
#
# forces use of invalid IP addresses. gethostbyaddr on some networks
# returns an invalid hostent entry even if the address is usable.
# FORCEIP must come before any used IP address in TARGET or HOST
# default OFF
FORCEIP         ON
#
#SERVICE        IP-ADDRESS              PORT    NAME(optional)
TARGET          146.75.124.23           18000   targetcomputer
#HOST           111.22.33.55            17000   SDT-host
# Target must be specified, host not. Port number is default, target
# has 18000 defined in tlayer.h
#
# setting USETIMEOUT to OFF disables timeouts and makes it possible
# to suspend the target or halt it at a C debugger breakpoint
# default ON
USETIMEOUT      OFF
#
# communication between host and service, only TCP (default) or TTYA/B (n/a)
#           NOTE, only TCP in this version
#COM            TARGET                  TCP
#
# in debug mode signals can be logged on file (default = stderr)
# Note! stderr will not work on Windows NT.
#
#LOG            dummylog.tmp
#
# debug on/off must be set to 'on' to activate logging.
#
#debug          on      #default = OFF
#
# If SDL diagram files are in a different directory than is hardcoded
# into the simulator (which assumes they are in the source directory
# which was specified in SDT when it was generated), you can explicitly
# give the path to the SDL source files here.
# Note! absolute path from root, i.e. starting with /dir1/dir2/... etc.
# trailing directory (a / in the end) is optional.
# If you can not get symbol trace on SDL editor, the SDLpath need to be
# defined to where you have your SDL source files
# Note! this only works on a flat directory structure.
#
#SDLpath        /home/kod/sdl/demongame
#
end
#
after an "end" statement host stops reading, so you can write anything!
```

Tarsim Shell (UNIX only)

On UNIX, there is a very basic shell included in `host.sct`.

If you start `host.sct` without the target running, you will enter a shell where only the commands **INIT** (try contacting target again) and **EXIT** (quit session) are applicable. If you know your target is running and still

do not get contact, please check that the specified IP address and PORT of the target in `tarsim.cfg` are correct. PORT is for the target hard-coded in `tlayer.h`, the header file for `tlayer.c`.

During simulation you can enter the tarsim shell by entering a "**:**" (colon). You can only leave the simulation if the simulator is not running with **GO** or **FOREVER**. If so, first stop the simulation with the *Break* button.

After entering the "**:**" you will see a welcome text to the shell. You can enter the command **HELP** to see the full command set, or **STATUS** to see the current simulation setup. You exit the shell by entering **EXIT**.

> **Note:**
>
> Be sure you are still in the tarsim shell when entering EXIT, since it is the same command for exiting the simulation. You can see if you are in the shell by the fact that all text begin with a colon as the first character in the text window. There is a time-out in the shell, after one minute of inactivity you get back to the simulator again.

LOG and **DEBUG** has the same syntax and meaning as in `tarsim.cfg`.

Addresses like TARGET and HOST, as well as the COM and SDLPATH statements, cannot be changed from the shell.

An additional facility for the logging is **GREP**, which gives a possibility to filter out up to 5 strings that you want to log and skip the rest.

To clear all GREP's and log everything:

```
GREP    *
```

To add a string (this example looks for MSCE, i.e. signals to the MSC Editor):

```
GREP    MSCE
```

Strings are not case sensitive by default. To search for the **exact** string `MyPath/demo.sdt`, prefix it with '=':

```
GREP    =MyPath/demo.sdt
```

If you want to log everything **except**, for instance, signals to the SDL Editor you can use the prefix '!':

```
GREP     !SDLE
```

Note:

If you have GREP MSC active and then enter GREP !MSC, status will be changed of the previously entered argument, so you cannot enter two contradictions and have both active.

You can also combine the inverse and exact functions:

```
GREP     !=MyPath/demo.sdt
```

Up to five GREP arguments can be active simultaneously and each can be up to 25 characters long.

If the GREP buffer is full, empty the old with GREP *.

You can empty the buffer and enter new argument in one statement:

```
GREP      *       MSCE
```

Executing and Terminating the Tools

When the simulator and gateway have been started, the simulator can be controlled from the Simulator UI on the host in the same way as an ordinary simulator. Differences may be seen in execution performance and in the printing of log messages. Using *Open* in the *File* menu in the SimUI, you specify the simulator file `host.sct`. After you have chosen this file, `host.sct` will try to establish contact with the target, according to settings in `tarsim.cfg`. This file must be in the same directory as `host.sct`, and the SDL Suite should also be started from this directory. (Setting Source Directory in the Organizer does not help SimUI find the target simulation file dependencies.)

The simulator on the target can be stopped by typing `<Ctrl+C>` on the console. It is though recommended only to stop the target by pressing the *Exit* button in the SimUI. The `host.sct` will then, before it finishes, send a stop signal to the target, and give it a chance to close all open file descriptors for TCP sockets. If not, you can get a bind error if you try to restart the target simulator without rebooting the target system.

The gateway on the host is terminated by issuing the command *Exit* or *Quit* in the SimUI.

When the simulator and gateway have been started, the simulator can be controlled from the Simulator UI on the host in the same way as an ordinary simulator. Differences may be seen in execution performance and in the printing of log messages. Using *Open* in the *File* menu in the SimUI, you specify the simulator file `host_smr.sct`. After you have chosen this file, `host_smr.sct` will try to establish contact with target, according to settings in `tarsim.cfg`. This file must be in the same directory as `host_smr.sct`, and the SDL Suite should also be started from this directory. (Setting Source Directory in the Organizer does not help SimUI find the target simulation file dependencies).

The simulator on the target can be stopped by typing `<Ctrl+C>` on the console. It is though recommended only to stop the target by pressing the *Exit* button in the SimUI. The `host_smr.sct` will then, before it finishes, send a stop signal to the target, and give it a chance to close all open file descriptors for TCP sockets. If not, you can get a bind error if you try to restart the target simulator without rebooting the target system.

The gateway on the host is terminated by issuing the command Exit or Quit in the SimUI.

On Windows NT the host gateway is named `host_smr.exe`.

The gateway are prebuilt but if needed they can be built using the makefiles host_smr.mak available in the platform directories.

Known Problems

- Some targets must be rebooted after a crash, since the OS does not make `fclose` on open file descriptors upon kill. You will see the error message “bind (0x30)”, meaning that bind fails since the target port address is busy. However, since enhanced shutdown and bind methods has been introduced, the risk of this has been reduced.
- Sometimes the SimUI interface hangs with a clock cursor after several reboots. Try the following solutions in the specified order until it works:

  1. Press the *Break* button in the SimUI.
  2. Select *Restart* in the *File* menu and restart the file `host.sct`

3. If you still get no access to the SimUI, try `<Ctrl+O>` (shortcut for open file) to reopen the file.

4. Find `host.sct` in the process list (`ps -ef` on Solaris) and kill it. Try 2-3 again.

5. Kill `sdtsimui` in the same way and restart it from the Organizer.

- If you get no contact with the target the first time, check the `tarsim.cfg` configuration file. Correct any incorrect IP address or port to the target (port is default 18000 except on VxWorks/VME) and restart `host.sct` like in step 2 above. The target's port is hard-coded into the file `tlayer.h`.

- If you cannot establish connection on an IP address and you still can ping/telnet the same address, try to set the command `FORCEIP ON` before any `TARGET` statement in `tarsim.cfg`. This forces use of an IP address even if it is reported invalid by `gethostbyaddr()`.

- On a slow network connection you might lose connection with the MSC editor. It can be impossible to start it by pressing the MSC button in the Simulator User Interface, or you can get the message "Error writing MSC event". Other problems that can occur is that it is impossible to `BREAK` when you are running the simulation with `GO` or `GO-FOREVER`.
  The solution to all of these problems is to set the environment variable `sdt_rpctimeout` to `100`. You need to restart the SDL Suite after doing this.

- If simulation starts but crashes after starting the MSC trace you are running out of stack space in the target. There is two solutions. Add the compiler flags MAX_READ_LENGTH=500 and MAX_WRI_LENGTH=500 in the makefile when you build the target. The flags reduces the default buffer size used by the simulation kernel. You may need to adjust the value after size and complexity of your system.
  The second fix is to set the stack size for the SDL task when you start it. Most RTOS:s has stack size as a parameter to taskSpawn, t_create or corresponding. Suitable stack size for remote simulation is 30-50 kB. Default stack in many RTOS kernels is 20kB, which will only work for very small systems.

- When using large messages between the simulator UI and the target simulator the buffers used for these messages maybe too small. The default size is 6000 bytes. To increase the buffers the define `DEFAULTBUFSIZE` in the source file `tlayer.h` can be changed. Note that for this to take effect you need to rebuild the gateway as well as the simulator.

Source Code Files

Source code for the target contains the complete standard Master Library. The only difference is that `post.o` does not contain any Postmaster, but a TCP socket communication program.

Both host and target use the same `tlayer.c` file, with the compile flags `-DTARGET` and `-DHOST`, respectively. Also the flag `-DTCP` need to be specified in both ends to get TCP functionality.

All other files in `sctworld.o` needs the flags `-DTARGETSIM`, `-D/compiler/`, `-D/rtos/`, `-DSCTDEBCLCOM` and `-DXMAIN_NAME=sdlmain`.

Also recommended: `-DMAX_READ_LENGTH=500` `-DMAX_WRI_LENGTH=500`

See the specific integrations for the RTOS and compiler specific flags.

There is also special flags for the TCP connection. Supported flags are currently VXWORKS_SDT, PSPS_SDT and WINDOWS. Default is standard Unix/Posix.

The source file structure in `tlayer.c` is also documented in the top of `tlayer.c` in a comment.

To make the target you will need `tlayer.c` and `tcp.c`, along with the `.h` files which are supplied. A makefile is also provided.

Building a remote simulation project

In the remsim directory there is a subdirectory called SampleProject Into the `INCLUDE` directory copy *.`c` and *.`h` from the installation under `sdt/sdtdir/{ARCH}sdtdir/INCLUDE` and `sdt/sdtdir/remsim/source`. The makefiles are referring to this directory. You could of course change the makefiles to refer to the original installation if you

do not need any changes.
In the directory REMSIMKERNEL you have the file Makefile.m to build a new kernel and make.opt for compiler settings used by both final build and making the kernel.
comp.opt defines the syntax for the generated makefile.
In make.opt you could by setting sctLD, sctLDFLAGS and sctEXTENSION decide if you want to build the final executable or a library to link into the final target build.

You build the kernel in the REMSIMKERNEL directory. The sample makefiles will work under Sun/Solaris with a GCC compiler. You only need to change for your cross compiler environment. Make the kernel by typing:

```
make -f makefile.m
```

The result will be a prelinked kernel called sctworld.o.

When you perform a make, you need to choose the option “Generate Makefile and use template” and select env.tpm in the REMSIMKERNEL directory. This links in the template sctenv.c (which in the delivery does nothing). You also need to select “Use kernel in directory” and select REMSIMKERNEL.
Please note that this only sample code to give you ideas how you can build your own cross compilation environment.

Special Information about Windows NT

Compiling

Both target and host has been tested on Windows NT 4.0 and compiled with Microsoft Visual C++ 5.0 and 6.0.

In the directory sdt/sdtdir/remsim/wini386, makefiles for Microsoft compilers are stored. Normally you will probably not run the target under Windows NT; this is just for demo purposes. Edit the top of the makefile, variables ccbindir and sdtsource to the absolute paths of your compiler binaries and the SDL Suite installation. Make from the prompt like this:
Microsoft: nmake /f ms_target.mak

The host part is called host_smr.exe and is located into the SimUI the same way as on UNIX.

Both parts must be compiled as 32 bit console applications.

Non-supported Features

Due to the fact that there are no stdout or stderr when you run a Windows GUI application, together with the difficulties to determine the right directory to write to a file in, all debugging and logging facilities has been excluded. That means that the entries DEBUG, GREP and LOG in `tarsim.cfg` will be ignored. There is no extra Target Simulation Shell as on UNIX (which is not needed when you cannot do any message logging). None of this affects the target simulation as such.

Other Known Limitations

It is considerably slower to do a remote simulation in Windows NT than on UNIX, even with the same system, same options, and running target and host on the same machine. There are ways to speed up the simulation. Recommended is to give **`set-tr 0`** as the first command in SimUI. Then all textual tracing is stopped. Instead you can ask for the value of certain variables of interest. MSC trace and SDL trace works OK; MSC is the quicker of the two.

You can of course do a normal host simulation on the NT machine and then you have no limitations. Target simulation is not recommended to be used more than for the final fine tuning into the target environment.

It is especially important in Windows to use the option `SDLPATH` in `tarsim.cfg` if the target has been compiled on another system. The parser in `host.exe` can read NFS syntax “/path/” and will convert it to Windows/DOS syntax “\path\”. The thing that cannot be fixed automatically is the correct logical drive letter. Hence you can specify where your SDL source files are with:

```
SDLPATH D:\path\sdlsource
```

in `tarsim.cfg`. Of course the source files must be the same as the ones the target was compiled from.

Chapter

52 *The SDL Explorer*

This chapter is a reference to the SDL Explorer user interface and a reference for the terminology used in validation.

For a guide to how to use the SDL Explorer, see chapter 53, *Validating a System*.

The SDL Explorer Monitor

An SDL Explorer generated by the SDL Suite is similar to a simulator in that it contains an interactive monitor system. The explorer monitor looks and behaves very similar to the simulator monitor; the only real difference is the set of available commands.

Monitor User Interfaces

Two different user interfaces are provided for the explorer monitor, a textual and a graphical.

The textual interface only allows commands to be entered from the keyboard, in the same way as for the simulator monitor.

The graphical interface still allows commands to be entered from the keyboard in the same way, but also provides buttons, menus and dialogs for easy access to commands and other features.

The textual interface is invoked by executing the generated explorer directly from the operating system prompt. This is called running an explorer in *textual mode*. When started, the explorer responds with the following text:

```
Welcome to the SDL EXPLORER

Command :
```

Another prompt may appear if the SDL system contains external synonyms. For more information, see "Supplying Values of External Synonyms" on page 2396 in chapter 53, *Validating a System*.

> **Note:**
>
> Before an explorer can be run in textual mode **on UNIX**, a command file must be executed from the operating system prompt. The file is called `telelogic.sou` or `telelogic.profile` and is located in the binary directory that is included in the user's path.
> For csh-compatible shells: **`source <bin dir>/telelogic.sou`**
> For sh-compatible shells: **`. <bin dir>/telelogic.profile`**

The graphical interface, known as the *SDL Explorer UI*, runs in a separate window. It is started from the Organizer by selecting *SDL > Explorer UI* from the *Tools* menu. The graphical interface is basically the same as for the simulator monitor. However, some special windows and

menus have been added; see "Graphical User Interface" on page 2349 for more information.

Activating the Monitor

Commands can be issued to the interactive monitor system when it becomes active. The explorer's monitor system becomes active:

- When the explorer is started.
- When the last command was Next or Random-Down and the transitions initiated by this command have completed.
- When the last command was Bit-State-Exploration, Verify-MSC or Exhaustive-Exploration and the behavior tree down to the defined search depth has been fully explored.
- When the last command was Random-Walk and the defined number of repetitions have been executed.
- Immediately after a report with the report action Abort has been generated during automatic state space exploration.
- In the Explorer UI, when the *Break* button is clicked in the *Explore* button module; in textual mode, when `<Return>` is pressed during an automatic state space exploration.

Note:

No other characters may be typed before `<Return>` is pressed.

Monitor Commands

This section provides an alphabetical listing of all available commands in the SDL Explorer monitor. The syntax of commands and data type literals in the explorer monitor is the same as for the simulator monitor. Commands may be abbreviated, as well as most command parameters. All input to the monitor is case insensitive. For more information, see "Syntax of Monitor Commands" on page 2132 and "Input and Output of Data Types" on page 2137 in chapter 49, *The SDL Simulator*.

? (Interactive Context Sensitive Help)

Parameters:

```
(None)
```

The monitor will respond to a '?' (question mark) by giving a list of all allowed values at the current position, or by a type name, when it is not suitable to enumerate the values. After the presentation of the list, the input can be continued.

? (Command Execution)

Parameters:

```
<Command name>
```

Executes the monitor command given as a parameter. This is a convenience function for the Explorer UI. Entering '?' as parameter gives a list of all possible command names.

Assign-Value

Parameters:

```
[ '(' <PId value> ')' ]
<Variable name> <Optional component selection>
<New value>
```

The new value is assigned to the specified variable in the process instance, service instance, or procedure given by the current scope. Sender, Offspring, and Parent may also be changed in this way, but their names may not be abbreviated.

After the command is given, the root of the behavior tree is set to the current system state.

It is, in a similar way as for the command Examine-Variable, possible to handle components in structured variables (struct, strings, array, and Ref) by appending the struct component name or a valid array index be-

fore the value to be assigned. Nested structs and arrays can be handled by entering a list of index values and struct component names.

If a PId is given within parenthesis, the scope is temporarily changed to this process instance instead.

Bit-State-Exploration

Parameters:

`(None)`

Starts an automatic state space exploration from the current system state using the bit state space algorithm. When the exploration is started, the following information is printed:

- Search depth: The maximum search depth of the exploration. This can be set with the command Define-Bit-State-Depth.
- Hash table size: The size of the hash table used during bit state exploration. This can be set with the command Define-Bit-State-Hash-Table-Size.

The exploration will continue until either the complete state space to the defined depth is explored, `<Return>` is pressed from the command prompt, or the *Break* button is pressed in the graphical user interface. The system is then returned to the state it was in before the exploration was started.

If a bit state exploration already has been started, but has been stopped, this command asks if the exploration should continue from where it was stopped, or restart from the beginning again.

A status message is printed for every 20,000 transitions that are executed.

When the exploration is finished or stopped, the Report Viewer is by default opened (this can be changed with the command Define-Report-Viewer-Autopopup). The following statistics are also printed:

- No of reports: The number of reported situations that may be listed with the List-Reports command.
- Generated states: The total number of system states generated.
- Truncated paths: The number of times the exploration reached the maximum depth, causing the current execution path to be truncated.

- Unique system states: The number of generated system states that are not duplicated anywhere in the behavior tree.
- Size of hash table: The size of the hash table in bits and bytes.
- No of bits set in hash table: The number of bits actually used to represent the generated state space.
- Collision risk: The risk, in percent, of a collision occurring in the hash table for two different system states. This would cause an incorrect truncation of an execution path in a newly generated state.
- Max depth: The maximum number of levels in the behavior tree that have been reached during the exploration.
- Current depth: The level of the behavior tree reached at the moment when the exploration was stopped. If it is -1, the exploration was completed, i.e., the complete behavior tree down to the specified depth was explored. If it is > 0, the exploration may be continued from this level by issuing the command again.
- Min state size: The smallest number of bytes used to store a system state when computing hash values.
- Max state size: The largest number of bytes used to store a system state when computing hash values.
- Symbol coverage: The percentage of the symbols in the process graphs that have been executed at least once.

Bottom

Parameters:

```
(None)
```

Go down in the behavior tree to the end of the current path.

Cd

Parameters:

```
<Directory>
```

Change the current working directory to the specified directory.

Channel-Disable

Parameters:

```
<Channel> | '-'
```

Disables all test values defined for all signals using the given channel. '-' means *all channels*. Use Channel-Enable to start using them again. See also Signal-Disable. Test values for a signal are only used if both the signal and the channel that transports the signal are enabled. By default, all signals and channels are enabled.

Channel-Enable

Parameters:

```
<Channel> | '-'
```

Enables all test values defined for all signals using the given channel. '-' means *all channels*. See also Signal-Enable. Test values for a signal are only used if both the signal and the channel that transports the signal are enabled. By default, all signals and channels are enabled.

Clear-Autolink-Configuration

Parameters:

```
<Directory>
```

Clears the current Autolink configuration.

Clear-Constraint

Parameters:

```
<Constraint name> | '-'
```

Removes the constraint specified by the parameter. If '-' is given instead of a constraint name, all constraints are cleared.

Note: Only constraints that are not used by any generated test case can be cleared. Constraints used by one or more test cases are cleared automatically if the corresponding test cases are cleared.

Clear-Coverage-Table

Parameters:

```
(None)
```

Clears the test coverage information.

Clear-Generated-Test-Case

Parameters:

```
<Test case name> | '-'
```

Removes the generated test case specified by the parameter from memory. However, the corresponding MSC test case on disk is preserved. If '-' is given instead of a test case name, all generated test cases are

cleared. Alternatively, several test cases may be selected by a pattern enclosed in apostrophes. Such a pattern may include the special characters '*', '?', and '[...]' and is evaluated just as in a UNIX command shell.

Clear-Instance-Conversion

Parameters:

(None)

Clears all defined instance conversions. See also the command Define-Instance-Conversion.

Note that this command will not change an already loaded MSC.

Clear-MSC

Parameters:

(None)

Clears the currently loaded MSC and sets the current state to the current root of the behavior tree.

Clear-MSC-Test-Case

Parameters:

<File name> | '-'

Removes the MSC test case specified by the parameter from disk. If '-' is given instead of a file name, all test cases are cleared. Alternatively, several test cases may be selected by a pattern enclosed in apostrophes. Such a pattern may include the special characters '*', '?', and '[...]' and is evaluated just as in a UNIX command shell.

Clear-MSC-Test-Step

Parameters:

<File name> | '-'

Removes the MSC test case specified by the parameter from disk. If '-' is given instead of a file name, all test steps are cleared. Alternatively, several test steps may be selected by a pattern enclosed in apostrophes. Such a pattern may include the special characters '*', '?', and '[...]' and is evaluated just as in a UNIX command shell.

Clear-Observer

Parameters:

<Process type>

Defines all process instances of the given type to be usual SDL processes, not observer processes.

Clear-Parameter-Test-Values

Parameters:

```
( <Signal> | '-' ) ( <Parameter number> | '-' )
( <Value> | '-' )
```

The test value described by the value parameter is cleared for the signal parameter given as the parameter to the command. If the specified value parameter is '-', clears all test values for the signal parameter given as the parameter to the command. If '-' is given instead of the parameter number then the test values for all parameters of the signal are cleared. If '-' is given instead of the signal name then all test values for all parameters to all signals are cleared.

Regenerates the set of signals that can be sent from the environment during state space exploration.

Clear-Reports

Parameters:

```
(None)
```

Delete the current reports from the latest state space exploration.

Clear-Rule

Parameters:

```
(None)
```

The currently defined rule is deleted.

Clear-Signal-Definitions

Parameters:

```
<Signal> | '-'
```

Clears all currently defined test values for the signal given by the parameter. If '-' is given, the test values for all signals are cleared.

> **Note:**
>
> The signals cleared by this command may be regenerated if any of the commands for defining test values for sorts or parameters are used.

Clear-Test-Values

Parameters:

```
( <Sort> | '-' ) ( <Value> | '-' )
```

Clears all test values for the sort given as parameter. If the specified sort parameter is '-', all test values for all sorts will be cleared. If the value parameter is '-', all test values for the specified sort will be cleared.

Regenerates the set of signals that can be sent from the environment during state space exploration.

Command-Log-Off

Parameters:

```
(None)
```

The command log facility is turned off; see the command Command-Log-On for details.

Command-Log-On

Parameters:

```
<Optional file name>
```

The command enables logging of all the commands given in the monitor. The first time the command is entered a file name for the log file has to be given as parameter. After that any further Command-Log-On commands, without a file name, will append more information to the previous log file, while a Command-Log-On command with a file name will close the old log file and start using a new file with the specified name.

Initially the command log facility is turned off. It can be turned off explicitly by using the command Command-Log-Off.

The generated log file is directly possible to use as a file in the command Include-File. It will, however, contain exactly the commands given in the session, even those that were not executed due to command errors. The concluding Command-Log-Off command will also be part of the log file.

Continue-Until-Branch

Parameters:

```
<Optional node number>
```

First go to the system state node which is the child with the specified number to the current system state. Same as the Next command. Then go down as long as there is only one child, i.e. a branch is found.

Continue-Up-Until-Branch

Parameters:

`(None)`

Go up in the behavior tree as long as there is only one child, i.e. a branch is found.

Default-Options

Parameters:

`(None)`

Resets all options in the Explorer to their default values and clears all reports. It also sets the current state to the current root. Compare with the command Reset.

Define-Autolink-Configuration

Parameters:

`<Configuration>`

Defines an Autolink configuration which might consist of:

1. Rules for the handling of constraints and signal parameter values (naming, replacement and parameterization).
2. Rules for the arrangement of test cases (test suite structure).
3. Rules for declaring ASP and PDU types.
4. A rule for stripping signal definitions from constraints.
5. Functions used by the rules 1 and 2 above.

If any rule or function is already defined, Autolink prompts the user to decide whether to add the new rules or to clear the existing Autolink configuration. If Define-Autolink-Configuration is part of a command file, the existing configuration is always replaced.

Rules beginning with `TRANSLATE` are evaluated when a constraint is created during test case generation. A translation rule specifies the name and the parameters of a constraint for a given SDL signal. In addition, it allows to replace signal parameter values by wildcards in a constraint

declaration table (TTCN matching mechanism). Translation rules can also be used to introduce test suite parameters and constants.

Test suite structure rules are evaluated when a test suite is saved. A test suite structure rule specifies a test group reference for test cases and/or test steps.

The two rules for the mapping of SDL sorts and signals onto ASP/PDU types are evaluated when writing the TTCN MP file. The same applies to the `STRIP-SIGNALS` rule that tells Autolink to create a constraint for a (single) parameter of an SDL signal instead for the signal itself.

An Autolink configuration has to obey the syntax rules described in "Autolink Configuration Syntax" on page 2382.

For a detailed description of the semantics, see "Syntax and Semantics of the Autolink Configuration" on page 1471 in chapter 35, *TTCN Test Suite Generation*.

Define-Autolink-Depth

Parameters:

```
<Depth>
```

Sets the maximum depth of the state space exploration for test case generation to the given value. The default value is 1000.

Define-Autolink-Generation-Mode

Parameters:

```
"Partial-Ordering" | "Total-Ordering"
```

Determines the way Autolink analyzes an MSC test case during test generation. If set to "Partial-Ordering", the standard semantics of MSC is used whereas "Total-Ordering" lets Autolink evaluate incoming and outgoing messages in an MSC straight from top to bottom. Typically, "Partial-Ordering" results in a behavior tree with several leaves whereas "Total-Ordering" produces only one sequence of events. The default setting is "Partial-Ordering". Note: The total ordering interpretation is not supported by the Generate-Test-Case command.

Define-Autolink-Hash-Table-Size

Parameters:

```
<Size>
```

Sets the size of the hash table used for test case generation to `<Size>` bytes. The default value is 1,000,000 bytes.

Define-Autolink-State-Space-Options

Parameters:

```
"On" | "Off"
```

Determines whether Autolink sets its own default state space options before generating test cases. If “On’, the state space options are automatically set when Generate-Test-Case is invoked and reset to the previous options when the command is completed. The default value is “On”.

Note that in general, the Explorer’s default options are too strict to find all inconclusive events. Also be aware that for Autolink to work correctly, the priority of “Input from ENV” has to be higher than the priority of all other event classes, and the queues for all channels to the environment have to be activated.

Define-Bit-State-Depth

Parameters:

```
<Depth>
```

The parameter is the maximum depth of bit state exploration, i.e., the number of levels to be reached in the behavior tree. If this level is reached during the exploration, the current path is8 truncated and the exploration continues in another node of the behavior tree. The default value is 100.

Define-Bit-State-Hash-Table-Size

Parameters:

```
<Size in bytes>
```

Sets the size of the hash table used to represent the generated state space during bit state exploration. The default value is 1,000,000 bytes.

Define-Bit-State-Iteration-Step

Parameters:

```
<Step>
```

The Bit-State-Exploration algorithm includes a feature to automatically make a number of explorations with an increased depth for each exploration. The iteration continues until the search depth is greater than the maximum depth search as defined by Define-Bit-State-Depth command or one exploration terminates without any truncations.

This command activates the feature and defines how much the depth is increased for each iteration. If <Step> is set to 0 the iterative exploration

is deactivated, otherwise <Step> defines how much the maximum depth is increased for each exploration.

The default value is 0, i.e. the feature is not activated.

Define-Channel-Queue

Parameters:

```
<Channel name> ( "On" | "Off" )
```

Adds or removes a queue for the specified channel. If a queue is added for a channel, it implies that when a signal is sent that is transported on this channel, it will be put into the queue associated with the channel. By default no channels have queues.

Define-Concurrent-TTCN

Parameters:

```
"On" | "Off"
```

Defines whether TTCN test suites should be saved in concurrent TTCN format.

The default value is “off”.

> **Note:**
>
> Concurrent TTCN is not supported in combination with the “Local” or “Inline” TTCN test steps format. If an error message appears when trying to enable concurrent TTCN, first set the test steps format to “Global” with Define-TTCN-Test-Steps-Format.

Define-Condition-Check

Parameters:

```
"On" | "Off"
```

Defines whether conditions in an MSC should be used for synchronization purposes during MSC verification and test case generation. If condition check is turned on, then events below a global condition can only occur if all events on all instances above that condition have been processed; the condition becomes a synchronization point. If the condition is not global, then only the connected instances are synchronized. If condition check is turned off, then conditions in MSCs are ignored.

The default value is “off”.

Note:

If condition check is turned on, then the semantics of conditions is different from the semantics given in the ITU-T Recommendation Z.120!

Define-Constraint

Parameters:

```
<Constraint name> <Signal definition>
```

Defines a new constraint, which is automatically written to the test suite if the user enters the Save-Test-Suite command later.

Define-Exhaustive-Depth

Parameters:

```
<Depth>
```

The parameter defines the depth of the search when performing exhaustive exploration. The default value is 100.

Define-Global-Timer

Parameters:

```
"On" | "Off"
```

Defines whether a global timer should be added to TTCN test suites. If this option is enabled, the following things are done automatically:

- A declaration for a timer “T_Global” is generated with default duration “PIX_T_Global” and a unit “s” (seconds);
- A test suite parameter “PIX_T_Global” with type “INTEGER” is declared;
- A statement “START T_Global” is inserted at the beginning of each Test Case Dynamic Behaviour table; if concurrent TTCN output is enabled, a “START T_Global” is also inserted at the beginning of each top-level Test Step Dynamic Behaviour table for all parallel test components;
- A statement “CANCEL T_Global” is appended to every test sequence in each Test Case Dynamic Behaviour table; if concurrent TTCN output is enabled, a “CANCEL T_Global” is also added to the end of every test sequence in each top-level Test Step Dynamic Behaviour table of all parallel test components.

The default value is “on”.

Note:

It is recommended that the decision on using global timers is made before test case generation is started and that it is not changed later. Changing the option after the first test case has been generated may lead to unexpected results.

The name of the global timer cannot be changed. However, the default duration and unit may be defined with the Define-Timer-Declaration command. This should be done before the first test case is generated.

Define-Instance-Conversion

Parameters:

```
<From string> <To string>
```

Adds an entry into an instance conversion table. If a string contains spaces, it must be quoted.

When reading instances in an MSC diagram and the name textually match a <From string> in the conversion table, it will be replaced by the <To string>. This is useful if you want to verify some, but not all, instances in an MSC, in which case you can convert the unwanted instances to be considered as environment.

Note that this command will not change an already loaded MSC.

Define-Integer-Output-Mode

Parameters:

```
"dec" | "hex" | "oct"
```

Defines whether integer values are printed in decimal, hexadecimal or octal format. In hexadecimal format the output is preceded with “0x”, in octal format the output is preceded with ‘0’ (a zero).

On input: if the format is set to hexadecimal or octal, the string determines the base as follows: After an optional leading sign a leading zero indicates octal conversion, and a leading “0x” hexadecimal conversion. Otherwise, decimal conversion is used.

The default is “dec”, and no input conversion is performed.

Define-Max-Input-Port-Length

Parameters:

```
<Number>
```

The maximum length of the input port queues is defined. If this length is exceeded during state space exploration, a report is generated (see "Max Queue Length Exceeded" on page 2368). The default value is 3.

Define-Max-Instance

Parameters:

```
<Number>
```

Defines the maximum number of instances allowed for any particular process type. If this number is exceeded during state space exploration, a report is generated (see "Create Errors" on page 2367). The default value is 100.

Define-Max-Signal-Definitions

Parameters:

```
<No of signals>
```

This command defines the maximum no of signals that will be added to the list of signals from the environment for any particular signal. The default value is 10.

Define-Max-State-Size

Parameters:

```
<Size in bytes>
```

Sets the size of an internal array used to store each system state when computing the hash value for the system state. The default size is 100,000 bytes.

Define-Max-Test-Values

Parameters:

```
<No of text values>
```

This command defines the maximum no of test values that will be generated for a particular data type or signal parameter. The default value is 10.

Define-Max-Transition-Length

Parameters:

```
<Number>
```

The maximum number of SDL symbols allowed to be executed during the performance of a behavior tree transition is defined. If this number is exceeded during state space exploration, a report is generated (see "Transition Length Error" on page 2372). The default value is 1,000.

Define-MSC-Ignore-Parameters

Parameters:

```
"On" | "Off"
```

Defines whether the parameters of signals (and timers) will be ignored (i.e. set to default values) when reading MSC diagrams. The default is "Off".

Note that this command will not change an already loaded MSC.

Define-MSC-Search-Mode

Parameters:

```
"Verification" | "Violation"
```

Defines options for MSC verification depending on the purpose of the exploration. "Verification" mode should be used if the intention with the exploration is to verify all MSC alternatives. "Violation" mode should be used if the intention is to check if the SDL system can violate the MSC.

The difference between "Verification" and "Violation" mode is that in "Verification" mode the explorer will:

- synchronize on all MSC starts and start of inline expressions, and
- perform a separate MSC action to enter an MSC / inline expression.

The synchronization in "Verification" mode implies that the explorer will try to execute all events in one MSC before continuing with the events in a subsequent MSC. The special MSC action is executed when all events in one MSC has been executed and it is time to start with the subsequent MSC. Note that if there are more than one subsequent MSC (e.g. due to an HMSC alternative or an inline expression) each MSC action will select one of the MSCs. In the trace output (and in the Navigator) the MSC actions will be presented as "Enter MSC XXX" where XXX is the name of the MSC that the explorer will try to execute next.

A simple way to get hands-on experience with the difference between "Verification" and "Violation" mode is to load an MSC that includes alternatives (either an HMSC or an MSC with inline expressions) using the Load-MSC command. Then set up the MSC search mode and use the Navigator to manually explore the state space.

Note that in "Verification" mode the **par** operator is not supported.

The default search mode is "Violation" mode.

Define-MSC-Test-Cases-Directory

Parameters:

```
<Directory name> | '-'
```

Defines the directory in which the MSC test cases are stored. If '-' is provided instead of a directory name, the test cases directory becomes undefined.

Define-MSC-Test-Steps-Directory

Parameters:

```
<Directory name> | '-'
```

Defines the directory in which the MSC test steps are stored. If '-' is provided instead of a directory name, the test steps directory becomes undefined.

Define-MSC-Trace-Action

Parameters:

```
"On" | "Off"
```

This command defines whether actions like tasks, decisions, etc. are shown in the MSC trace. The default value is "Off".

Define-MSC-Trace-Autopopup

Parameters:

```
"On" | "Off"
```

This command defines whether an MSC Editor automatically will pop up and show an MSC trace when going to a report. The default value is "On".

Define-MSC-Trace-State

Parameters:

```
"On" | "Off"
```

This command defines if the SDL process graph states are shown in the MSC trace. If "On", a condition symbol will be shown in the MSC trace each time a process performs a nextstate. The default value is "On".

Define-MSC-Trace-Channels

Parameters:

```
"On" | "Off"
```

Defines whether the env instance should be split into one instance for each channel connected to env in the MSC trace. The default is "Off".

Define-MSC-Verification-Algorithm

Parameters:

```
"TreeSearch" | "BitState"
```

Defines the search algorithm that is used for MSC verification. Two options are available, the tree search algorithm and the bit state algorithm. The default is “BitState”.

Note that this command only defines what search algorithms that is used by the Verify-MSC command. It is possible to use other search algorithms, like Exhaustive-Exploration, by manually loading the MSC with the Load-MSC command and then starting the desired search algorithm.

Define-MSC-Verification-Depth

Parameters:

```
<Depth>
```

The parameter defines the depth of the search when performing MSC verification using the Verify-MSC command. The default value is 1,000.

Define-Observer

Parameters:

```
<Process type>
```

Defines all process instances of the given type to be observer processes.

Define-Parameter-Test-Value

Parameters:

```
<Signal> <Parameter number> <Value>
```

The parameter test value described by the command parameters is added to the current set of test values. The list of signals that can be sent from the environment is regenerated based on the new set of test values.

Define-Priorities

Parameters:

```
<Internal events> <Input from ENV> <Timeout events>
<Channel output> <Spontaneous transitions>
```

Defines the priorities for the different event classes. The priorities can be set individually to 1, 2, 3, 4 or 5. For more information about event classes and priorities, see “Event Priorities” on page 2467 in chapter 53, *Validating a System*.

The default priorities are:
- Internal events: 1
- Input from ENV: 2
- Timeout events: 2
- Channel outputs: 1
- Spontaneous transitions: 2

Note that the priorities of events processed in the explorer also is affected by the command "Define-Symbol-Time" on page 2320.

Define-Random-Walk-Depth

Parameters:

```
<Depth>
```

The parameter defines the depth of the search when performing random walk exploration. The default value is 100.

Define-Random-Walk-Repetitions

Parameters:

```
<No of repetitions>
```

The parameter defines the number of times the search is performed from the start state when performing random walk exploration. The default is 100.

Define-Report-Abort

Parameters:

```
<Report type> | '-'
```

Defines that the state space exploration will be aborted whenever a report of the specified type is generated.

The available report types are listed with the command Show-Options. They are also described in "Rules Checked During Exploration" on page 2366. If the parameter is specified as '-', all report types will be defined in the same way.

Define-Report-Continue

Parameters:

```
<Report type> | '-'
```

Defines that a state space exploration will continue past a state where a report of the specified type is generated. With this definition, the exploration of the behavior tree is not affected by a report being generated.

The available report types are listed with the command Show-Options. They are also described in "Rules Checked During Exploration" on page 2366. If the parameter is specified as '-', all report types will be defined in the same way.

Define-Report-Log

Parameters:

```
<Report type> ( "Off" | "One" | "All" )
```

Defines how many reports of a specified type will be stored in the list of found reports when they are encountered during state space exploration. Default for all report types is "One".

If "Off" is specified for a report type, reports of this type will not be stored, which implies for example that they will not be listed by the List-Reports command. However, the reports will be generated and the appropriate action taken as specified by the commands Define-Report-Abort, Define-Report-Continue and Define-Report-Prune.

If "One" is specified only one occurrence of each reported situation is stored.

If "All" is specified then all occurrences of a reported situation that have different execution paths is stored in the list.

The available report types are listed with the command Show-Options. They are also described in "Rules Checked During Exploration" on page 2366. If the parameter is specified as '-', all report types will be defined in the same way.

Define-Report-Prune

Parameters:

```
<Report type> | '-'
```

Defines that a state space exploration will not continue past a state where a report of the specified type is generated. Thus, the part of the behavior tree beneath the state will not be explored. Instead, the exploration will continue in the siblings or parents of the state. This is also known as "pruning" the behavior tree at the state. This is the default behavior for all report types.

The available report types are listed with the command Show-Options. They are also described in "Rules Checked During Exploration" on page 2366. If the parameter is specified as '-', all report types will be defined in the same way.

Define-Report-Viewer-Autopopup

Parameters:

```
"On" | "Off"
```

This command defines whether the Report Viewer automatically will pop up when an automatic exploration is finished. The default is “On”.

Define-Root

Parameters:

```
"Original" | "Current"
```

Defines the root of the behavior tree to be either the current system state or the original start state of the SDL system.

> **Note:**
>
> When the root is redefined, all paths, e.g. MSC traces or report paths, will start in the new root, not in the original start state.

Define-Rule

Parameters:

```
<User-defined rule>
```

A new rule is defined that will be checked during state space exploration.

Define-Scheduling

Parameters:

```
"All" | "First"
```

Defines which process instances in the ready queue are allowed to execute at each state. The parameter defines the scheduling as follows:

- `All`
  All process instances in the ready queue are allowed to execute at each state.
- `First`
  Only the first process instance in the ready queue is allowed to execute at each state.

The default is “First”.

Define-Signal

Parameters:

```
<Signal> <Optional parameter values>
```

A signal that is to be sent to the SDL system from the environment is defined. The signal is defined by its name and optionally the values of its parameters. Multiple Define-Signal commands may be used to define the same signal, but with different values for the parameters.

> **Note:**
>
> The signals defined by this command will be destroyed if the signals are regenerated, i.e., if any of the commands defining test values for sorts or signal parameters are used.

Define-Spontaneous-Transition-Progress

Parameters:

```
"On" | "Off"
```

Defines whether a spontaneous transition (input none) is considered as progress when performing non-progress loop check. Default is that spontaneous transition is considered to be progress, i.e. “On”. See “Non Progress Loop Error” on page 2372.

Define-Symbol-Time

Parameters:

```
"Zero" | "Undefined"
```

The time it takes to execute one symbol, e.g. an input, task or decision, in an SDL process is defined either to be zero or undefined. If it is set to zero, it is assumed that all actions performed by process instances take are infinitely fast compared to the timer values that are used in the system. If the symbol time is set to undefined, no assumption is made about how long time it takes for processes to execute symbols. Consider for example a situation where a process sets a timer with a duration 5 and then executes something that may take a long time, e.g. a long loop, and the sets a timer with duration 1. If symbol time is set to zero then the second timer will always expire first. If symbol timer is set to undefined then both timers can potentially expire first.

Note that when symbol time is set to zero, no timer will expire if an internal action is possible, even if internal and timer events have the same priority as set by the command Define-Priorities.

The default value of the symbol time options is “Zero”.

Define-Test-Value

Parameters:

```
<Sort> <Value>
```

The test value described by the parameters is added to the current set of test values. The list of signals that can be sent from the environment is regenerated based on the new set of test values.

> **Note:**
>
> When regenerating the set of signals, all signals that have been manually defined using the Define-Signal command will be lost.

Define-Timer-Check-Level

Parameters:

```
<Number>
```

The action taken during MSC verification when checking timer statements (set, reset and timeout) is defined. The possible values for the specified number are:

- 0: No checking of timer events is performed.
- 1: If a timer event exists in the MSC a matching timer event must exist in the explored SDL path, but a timer event in the explored SDL path is accepted even if there is no corresponding MSC timer event.
- 2: All timer events in the MSC must match a corresponding timer event in the explored SDL path, and vice versa.

The default value is 1.

Define-Timer-Declaration

Parameters:

```
<Timer name> <Duration> <Unit>
```

Creates or updates a TTCN timer declaration. Autolink distinguishes between implicit timer declarations, which are created from test description MSCs during test case generation, and explicit timer declarations, which are created with this command. Valid explicit declarations always override implicit ones. Existing explicit declarations cannot be updated.

The duration may be an integer value or a valid test suite parameter identifier. If a test suite parameter is specified, a declaration for this pa-

rameter is generated as well. If no valid duration is specified, the duration field in the declaration remains empty.

The unit must be a valid TTCN unit: “ps”, “ns”, “us”, “ms”, “s” or “min”. No declaration is made if an invalid unit is specified.

Note:

It is recommended that all timers appearing in MSC test cases and test steps are declared explicitly prior to test case generation.

Define-Timer-Progress

Parameters:

```
"On" | "Off"
```

Defines if the expiration of a timer is considered as progress when performing non-progress loop check. Default is that timer expiration is considered to be progress, i.e. “On”. See “Non Progress Loop Error” on page 2372.

Define-Transition

Parameters:

```
"SDL" | "Symbol-Sequence"
```

Defines the semantics and length of the transitions in the behavior tree. The parameter defines the transitions as follows:

- `SDL`
  The behavior tree transitions correspond to complete SDL process graph transitions.
- `Symbol-Sequence`
  The behavior tree transitions correspond to the longest sequence of SDL symbols that can be executed without any interaction with other process instances.

The default is “SDL”.

Define-Tree-Search-Depth

Parameters:

```
<Depth>
```

Defines the maximum search depth for tree search.

The default value is 100.

Define-TTCN-Compatibility

Parameters:

"Standard" | "ITEX"

Determines whether Autolink generates the declarations part of a TTCN test suite. If set to "Standard", Autolink creates declarations with ASN.1 data field names being constructed of the form "data type+parameter number" (e.g. "Integer2"). If set to "ITEX", no declarations are created. In this case, ASN.1 data field names used in constraint definitions are represented by '#' characters. This allows to merge the test suite with declarations generated by Link in the TTCN Suite. The default value is "Standard".

This command exists for backward compatibility reasons only.

Define-TTCN-Signal-Mapping

Parameters:

"ASP" | "PDU"

Determines whether SDL signals are mapped onto ASN.1 ASP constraints or ASN.1 PDU constraints. By default, signals are mapped onto ASPs. The setting of Define-TTCN-Signal-Mapping can be overruled for single signals by the use of the 'ASP-Types' and 'PDU-Types' commands in the Autolink configuration language (cf. Define-Autolink-Configuration).

Define-TTCN-Test-Steps-Format

Parameters:

"Global" | "Local" | "Inline"

Defines how test steps are written in a TTCN test suite. "Global" is the default value.

If set to "Global", each test step is saved in a separate behaviour description table. If a test step is used in more than one test case, then only one behaviour description table is generated.

If set to "Local", each test step is saved as a local tree in the behaviour table of the corresponding test case. If a test step is used several times in a single test case, then only one behaviour description is generated.

If set to "Inline", all test step trees are directly inserted into the test case descriptions.

Note that a test step which is used in several places may lead to trees with different inconclusive events or different verdicts. In this case, the test steps are stored in separate tables with unique names.

> **Note:**
>
> Test step formats “Local” and “Inline” are not supported in combination with concurrent TTCN. If an error message appears when trying to change the test steps format, first disable concurrent TTCN with Define-Concurrent-TTCN.

Define-Variable-Mode

Parameters:

```
<Process or Process Type> [ <Variable name> |
“Parent” | “Offspring” | “Sender” ]
[ “Compare” | “Skip” ]
```

This command defines how a specific variable is treated when comparing two system states during state space exploration. If the value for a variable is “Compare”, this variable will be taken into account when comparing two system states. If the value is “Skip”, this variable will not be taken into account, i.e. if the only difference between two system states is that values of variables in “Skip” mode differs, then the system states will be considered equal.

The purpose of the “Skip” mode for variables is to optimize the state space search. There are two different situations where this command can be used:

- All variables that is known to be constant during an exploration can be declared “Skip”.
- All variables that will not have an effect on the dynamic behavior of the system, i.e. that will not affect the path through a decision or the expression in an “output to”, can be declared “Skip”.

The benefit with constant variables in “Skip” mode is that the Explorer will ignore these variables when computing hash values. This can for large data structures like arrays mean that the performance of the explorer can be considerably improved.

Detailed-Exa-Var

Parameters:

```
(None)
```

When printing structs containing components with default value, these values are explicitly printed after this command is given.

Down

Parameters:

```
<Number of levels>
```

Go down the specified number of levels in the behavior tree, each time selecting the child of the current system state that is part of the current path. If the parameter is too large, Down will stop at the end of the current path.

Evaluate-Rule

Parameters:

```
(None)
```

The currently defined rule is evaluated with respect to the current system state. The command prints whether or not the rule is satisfied.

Examine-Channel-Signal

Parameters:

```
<Channel name> <Entry number>
```

The parameters of the signal instance at the position equal to the entry number in the queue of the specified channel are printed. The entry number is the number associated with the signal instance when the command List-Channel-Queue is used.

Examine-PId

Parameters:

```
(None)
```

Information about the process instance given by the current scope is printed (see the Set-Scope command for an explanation of scope). This information contains the current values of Parent, Offspring, Sender and a list of all currently active procedure calls made by the process instance. The list starts with the latest procedure call and ends with the process instance itself.

If the process instance contains service(s), information about all services is also printed.

Examine-Signal-Instance

Parameters:

```
<Entry number>
```

The parameters of the signal instance at the specified position in the input port of the process instance given by the current scope are printed (see the Set-Scope command for an explanation of scope). The entry number is the number associated with the signal instance when the command List-Input-Port is used.

Examine-Timer-Instance

Parameters:

```
<Entry number>
```

The parameters of the specified timer instance are printed. The entry number is the number associated with the timer when the List-Timer command is used.

Examine-Variable

Parameters:

```
[ '(' <PId value> ')' ]
<Optional variable name>
<Optional component selection>
```

The value of the specified variable or formal parameter in the current scope is printed (see the Set-Scope command for an explanation of scope). Variable names may be abbreviated. If no variable name is given, all variable and formal parameter values of the process instance given by the current scope are printed. Sender, Offspring, and Parent may also be examined in this way. Their names, however, may not be abbreviated and they are not included in the list of all variables.

> **Note:**
>
> If a variable is exported, both its current value and its exported value are printed.

It is possible to examine only a component of a struct, string or array variable, by appending the struct component name or a valid array index value as an additional parameter. The component selection can handle structs and arrays within structs and arrays to any depth by giving a list component selection parameters. SDL syntax with '!' and "( )" as well as just spaces, can be used to separate the names and the index values.

It is also possible to print a range of an array by giving “FromIndex : ToIndex” after an array name. Note that the space before the ‘:’ is required if FromIndex is a name (enumeration literal), and that no further component selection is possible after a range specification.

To see the possible components that are available in the variable, the variable name must be appended by a space and a ‘?’ on input. A list of components or a type name is then given, after which the input can be continued. After a component name, it is possible to append a ‘?’ again to list possible sub components.

To print the value of the data referenced by a Ref pointer it is possible to use the SDL syntax, i.e. the “*>” notation. If, for example, `Iref` is a Ref(Integer) variable, then `Iref*>` is the integer referenced by this pointer. If `Sref` is a Ref of a struct, then `Sref*> ! Comp1` is the Comp1 component in the referenced struct. The sequence `*> !` can in the monitor be replaced by `->` (as for example in C).

If a PId is given within parenthesis, the scope is temporarily changed to this process instance instead.

Exhaustive-Exploration

Parameters:

`(None)`

Starts an automatic state space exploration from the current system state, where the entire generated state space is stored in primary memory. This is only recommended for SDL systems with small state spaces.

The exploration will continue until either the complete state space to the defined depth is explored, `<Return>` is pressed from the command prompt, or the *Break* button is pressed in the graphical user interface. The system is then returned to its initial system state. The maximum depth of the exploration can be set with the command Define-Exhaustive-Depth.

If an exhaustive exploration already has been started, but has been stopped, this command asks if the exploration should continue from where it was stopped, or restart from the beginning again.

A status message is printed every 50,000 states that are generated. When the exploration is finished or stopped, the same information as for a bit state exploration is printed.

Exit

Parameters:

(None)

The executing explorer is terminated. If the command is abbreviated, the monitor asks for confirmation. If any of the Explorer options have been changed, the monitor will ask if the changed options should be saved. If so, the changed options are saved in a file `.valinit` **(on UNIX)**, or `valinit.com` **(in Windows)**, in the directory from where the SDL Suite was started. This file is automatically loaded the next time a Explorer is started from the same directory, thus restoring the previously saved options.

This is the same command as Quit.

Extract-Signal-Definitions-From-MSC

Parameters:

<Directory name> (<File name> | '-')

Extracts all signals from an MSC which are to be sent from the environment into the SDL system and adds them to the list of active signals. This command is only applicable to basic MSCs in textual format, i.e. to those MSCs stored with file extension `.mpr`. If '-' is given instead of a file name, signal definitions from all MSCs in the specified directory are extracted. Alternatively, several MSCs may be selected by a pattern enclosed in apostrophes. Such a pattern may include the special characters '`*`', '`?`', and '`[...]`' and is evaluated just as in a UNIX command shell.

Generate-Test-Case

Parameters:

<File name> | '-'

Starts the state space search algorithm which constructs an internal data structure for the MSC test case specified by the parameter. Additionally, constraints for all signals occurring in the test case are generated. At the end all constraints with identical signal definitions are merged.

If the command parameter is specified as '`-`', test cases are generated for all MSC test cases in the current test cases directory. Alternatively, several MSC test cases may be selected by a pattern enclosed in apostrophes. Such a pattern may include the special characters '`*`', '`?`', and '`[...]`' and is evaluated just as in a UNIX command shell.

If the previous generation of a test case has been interrupted, Autolink prompts the user to decide whether to continue the interrupted generation or to abort it. No file name is needed if the test case generation is to be continued.

Note that this command requires a TTCN Link license in addition to the Explorer license.

Goto-Path

Parameters:

```
<Path>
```

Go to the system state specified by the path. For details about paths, see the command Print-Path.

Goto-Report

Parameters:

```
<Report number>
```

Go to the state in the behavior tree where the report with the corresponding number has been found. The last behavior tree transition that was executed before the reported situation is printed, with the same information as for a full trace during simulation. The report number is the number associated with the report when the command List-Reports is used. If only one report exists, the report number is optional.

Help

Parameters:

```
<Optional command name>
```

Issuing the Help command without a parameter will print all the available commands. If a command name is given as parameter, this command will be explained.

Include-File

Parameters:

```
<File name>
```

This command provides the possibility to execute a sequence of monitor commands stored in a text file. The Include-File facility can be useful for including, for example, an initialization sequence or a complete test case. It is allowed to use Include-File in an included sequence of commands; up to five nested levels of include files can be handled.

List-Channel-Queue

Parameters:

```
<Channel name>
```

A list of all signal instances in the specified channel queue is printed. For each signal instance an entry number, the signal type, and the sending process instance is given. The entry number can be used in the command Examine-Channel-Signal.

List-Constraints

Parameters:

```
(None)
```

Lists all currently defined constraints.

List-Generated-Test-Cases

Parameters:

```
(None)
```

Lists all test cases which have been generated either by Generate-Test-Case or Translate-MSC-Into-Test-Case, or which have been loaded with the Load-Generated-Test-Cases command.

List-Input-Port

Parameters:

```
[ '(' <PId value> ')' ]
```

A list of all signal instances in the input port of the process instance given by the current scope is printed (see the Set-Scope command for an explanation of scope). For each signal instance an entry number, the signal type, and the sending process instance is given. A '*' before the entry number indicates that the corresponding signal instance is the signal instance that will be consumed in the next transition performed by the process instance. The entry number can be used in the command Examine-Signal-Instance.

If a PId is given within parenthesis information about this process instance is printed instead.

List-Instance-Conversion

Parameters:

```
(None)
```

Lists all defined instance conversions. See also the command Define-Instance-Conversion.

List-MSC-Test-Cases-And-Test-Steps

Parameters:

```
(None)
```

Lists all MSC test cases and test steps currently stored in the test cases and test steps directories.

List-Next

Parameters:

```
(None)
```

A list of the possible behavior tree transitions that can follow from the current system state is printed.

> **Note:**
>
> The number of possible transitions depends on the selected state space options.

List-Observers

Parameters:

```
(None)
```

List all process types which are defined to be observer processes.

List-Parameter-Test-Values

Parameters:

```
(None)
```

Lists all currently defined test values for signal parameters.

List-Process

Parameters:

```
<Optional process name>
```

A list of all process instances with the specified process name is printed. If no process name is specified all process instances in the system are listed. The list will contain the same details as described for the List-Ready-Queue command.

If the process contains services, information about the currently active service is also printed. To get information about all services use the Examine-PId command.

List-Ready-Queue

Parameters:

```
(None)
```

A list of process instances in the ready queue is printed. For more information, see the Simulator command "List-Ready-Queue" on page 2157 in chapter 49, *The SDL Simulator*.

List-Reports

Parameters:

```
(None)
```

All situations that have been reported during state space exploration are printed. For each report, the error or warning message describing the situation is printed, together with the depth in the behavior tree where it first occurred. Only one occurrence of each reported situation is printed; the one with the shortest path from the root of the behavior tree. The report numbers printed can be used in the command Goto-Report.

List-Signal-Definitions

Parameters:

```
(None)
```

A list of all currently defined signals is printed.

List-Test-Values

Parameters:

```
(None)
```

Lists all test values that currently are defined.

List-Timer

Parameters:

```
(None)
```

A list of all currently active timers is printed. For each timer, its corresponding process instance and associated time is given. An entry number will also be part of the list, which can be used in the command Examine-Timer-Instance.

Load-Constraints

Parameters:

```
<File name>
```

Loads all constraints stored in the file specified by the parameter. The suggested file extension is `.con`.

Load-Generated-Test-Cases

Parameters:

```
<File name>
```

Loads all generated test cases stored in the file specified by the parameter. The suggested file extension is `.gen`.

Load-MSC

Parameters:

```
<MSC file name>
```

The specified Message Sequence Chart is loaded into the Explorer. Loading an MSC always resets the state space and sets the current state to the root of the behavior tree.

Load-Signal-Definitions

Parameters:

```
<File name>
```

A command file with Define-Signal commands is loaded and the signals are defined. This command exists for backward compatibility reasons only.

Log-Off

Parameters:

```
(None)
```

The command Log-Off turns off the interaction log facility, which is described in the command Log-On.

Log-On

Parameters:

```
<Optional file name>
```

The command Log-On takes an optional file name as a parameter and enables logging of all the interaction between the Explorer and the user that is visible on the screen. The first time the command is entered, a file name for the log file has to be given as parameter. After that any further Log-On commands, without a file name, will append more information to the previous log file, while a Log-On with a file name will close the old log file and start using a new file with the specified file name.

Initially the interaction log facility is turned off. It can be turned off explicitly by using the command Log-Off.

Merge-Constraints

Parameters:

```
<First name> <Second name>
```

Merges the two constraints specified by the parameters. The resulting constraint has at least the formal parameters of the original constraints. If additional constraint parameters are necessary (e.g. when merging two constraints with signals S(1,1) and S(1,2)), Autolink prompts the user to enter a name for each new formal parameter. Merge-Constraints can only be applied to constraints with the same signal. The new constraint gets the name of the second constraint.

Merge-Report-File

Parameters:

```
<File name>
```

An existing report file is opened and the reports in it are added to the current reports.

MSC-Log-File

Parameters:

```
<File name>
```

A log file is produced containing information about the Message Sequence Chart for the trace from the root of the behavior tree to the current system state. This file can be opened in an MSC Editor; we recommend using a filename with the suffix `.mpr`, since that suffix is used as default.

MSC-Trace

Parameters:

```
(None)
```

An MSC Editor is opened, showing the Message Sequence Chart for the current execution path. The current state is shown in the MSC by the selection of some of the symbols. Whenever the current state and/or current path changes in the Explorer, the MSC will be updated.

If the MSC trace is already on, this command will turn it off.

New-Report-File

Parameters:

```
<File name>
```

A new report file for report storage is created. The current reports are deleted.

Next

Parameters:

```
<Transition number>
```

Go to a system state in the next level of the behavior tree, i.e. a child to the current system state. The parameter is the transition number given by the List-Next command.

Open-Report-File

Parameters:

```
<File name>
```

An existing report file is opened and the reports in it are loaded. The current reports are deleted.

Parameterize-Constraint

Parameters:

```
<Constraint name> <Replacement definition>
```

Replaces concrete values in a constraint with formal parameters. The replacement is defined by a sequence of pairs of a signal parameter number followed by the corresponding formal parameter name. The definition must be terminated with a 0.

Print-Autolink-Configuration

Parameters:

```
(None)
```

Displays the current Autolink configuration on the screen.

Print-Evaluated-Rule

Parameters:

```
(None)
```

The currently defined rule is printed with the values obtained from the last evaluation of the rule. The printed information is in the form of a so-called parse tree, and may require some knowledge of such structures to be interpreted correctly.

Print-File

Parameters:

```
<File name>
```

The content of the named text file is displayed on the screen.

Print-Generated-Test-Case

Parameters:

```
<Test case name> | '-'
```

Displays the internal representation of the generated test case specified by the parameter on the screen. If '-' is given instead of a test case name, all generated test cases are shown. Alternatively, several test cases may be selected by a pattern enclosed in apostrophes. Such a pattern may include the special characters '*', '?', and '[...]' and is evaluated just as in a UNIX command shell.

Print-MSC

Parameters:

```
(None)
```

A textual version of the currently loaded MSC is printed. The format is similar to the textual MSC format defined in ITU recommendation Z.120.

Print-Path

Parameters:

```
(None)
```

The path to the current system state is printed. A path is a sequence of integer numbers, terminated with a 0, describing how to get to the current system state. The first number indicates what transition to select from the root, the second number what transition to choose from this state, etc. To go to the state specified by a path, use the command Goto-Path.

Print-Report-File-Name

Parameters:

```
(None)
```

The name of the current report file is printed.

Print-Rule

Parameters:

```
(None)
```

The currently defined rule is printed.

Print-Trace

Parameters:

```
<Number of levels>
```

The textual trace leading to the current system state is printed. The same information as for a full trace during simulation is printed for each behavior tree transition. The parameter determines how many levels up the trace should start. For example, Print-Trace 10 will print the ten last transitions.

Quit

Parameters:

```
(None)
```

The executing explorer is terminated. If the command is abbreviated, the monitor asks for confirmation. If any of the Explorer options have been changed, the monitor will ask if the changed options should be saved. If so, the changed options are saved in the file `.valinit` **(on UNIX)**, or `valinit.com` **(in Windows)**, in the directory from where the SDL Suite was started. This file is automatically loaded the next time a Explorer is started from the same directory, thus restoring the previously saved options.

This is the same command as Exit.

Random-Down

Parameters:

```
<Number of levels>
```

Go down the specified number of levels in the behavior tree, each time selecting a random child of the current system state.

Random-Walk

Parameters:

```
(None)
```

This command will perform an automatic exploration of the state space from the current system state. Random walk is based on the idea that if more than one transition is possible in a particular system state, one of them will be chosen at random. When the exploration is started, the following information is printed:

- Search depth: The maximum depth to walk down. This can be set with the command Define-Random-Walk-Depth.

- Repetitions: The number of random walks to perform from the start state. This can be set with the command Define-Random-Walk-Repetitions.

The exploration will continue until either it is finished, `<Return>` is pressed from the command prompt, or the *Break* button is pressed in the graphical user interface. The system is then returned to the state it was in before the exploration was started.

When the exploration is finished or stopped, the Report Viewer is by default opened (this can be changed with the command Define-Report-Viewer-Autopopup). A few statistics are also printed; see the command Bit-State-Exploration for an explanation of these.

Rename-Constraint

Parameters:

`<Old name> <New name>`

Renames a constraint. If a constraint with the same "new" name but a different signal definition already exists, the user is prompted to choose between one of the following operations (which can be abbreviated):

- `Rename`: A different name can be chosen for the constraint.
- `Keep_old`: The definition of the renamed constraint is cleared. Any references from previously generated test cases to the renamed constraint are redirected to the old constraint.
- `Overwrite_old`: The definition of the old constraint is cleared. Any references from previously generated test cases to the old constraint are redirected to the renamed constraint.

Restriction: If one of the constraints contains formal parameters, then a different name must always be chosen for the renamed constraint. Rename-Constraint in combination with its options `Keep_old` and `Overwrite_old` allows to merge two constraints, e.g. if some signal parameters are irrelevant (see also the command Merge-Constraints and consider the use of `CONSTRAINT MATCH` ... statements in an Autolink configuration).

Reset

Parameters:

`(None)`

Resets the state of the Explorer to its initial state. This command resets all options and test values in the Explorer to their initial values and clears all reports, MSCs, and user-defined rules. It also sets the current state and the current root to the original start system state.

This command reads the `.valinit` **(on UNIX)**, or `valinit.com` **(in Windows)**, file (see the command Exit). It is equivalent to closing the Explorer and starting it again. Compare with the command Default-Options.

Save-As-Report-File

Parameters:

```
<File name>
```

The current reports are saved in a new file. The name of the current report file is set to the new file.

Save-Autolink-Configuration

Parameters:

```
<File name>
```

Saves the current Autolink configuration in a command file. The command name `Define-Autolink-Configuration` and the terminating `End` are written at the beginning and the end of the file in order to allow to reload the configuration with Include-File. The suggested file extension is `.com`.

Save-Constraint

Parameters:

```
( <Constraint name> | '-' ) <File name>
```

Saves the constraint in a file identified by `<File name>`. If '-' is given instead of a constraint name, all currently defined constraints are saved. The suggested file extension is `.con`.

Save-Coverage-Table

Parameters:

```
<File name>
```

Test coverage information is saved in the specified file. The test coverage table consists of two parts, a Profiling Information section, and a Coverage Table Details section. This is the same type of file as generated by the Simulator; for more detailed information about the file, see

"Print-Coverage-Table" on page 2163 in chapter 49, *The SDL Simulator*.

Save-Error-Reports-As-MSCs

Parameters:

```
(None)
```

Saves all reports (except those indicating an MSC verification) as MSCs in the target directory. The reports are stored as complete MSCs with file extension `.mpr`.

For each MSC a generic name consisting of the name of the currently loaded MSC (or "ErrorReport", if none is loaded), a four letter abbreviation of the error type and a sequence number is generated.

Save-Error-Reports-As-MSCs helps to find out what went wrong when running a batch job with a large number of MSC verifications.

Note: This command does not generate MSCs for MSC violation reports if the current MSC could be verified.

Save-Generated-Test-Case

Parameters:

```
( <Test case name> | '-' ) <File name>
```

Saves the generated test case in a file. If '-' is given instead of a test case name, all generated test cases which are currently stored in memory are saved. Alternatively, several test cases may be selected by a pattern enclosed in apostrophes. Such a pattern may include the special characters '`*`', '`?`', and '`[...]`' and is evaluated just as in a UNIX command shell.The suggested file extension is `.gen`.

This command allows to compute test cases in parallel on different machines and to merge them afterwards by using Load-Generated-Test-Cases.

Save-MSC-Test-Case

Parameters:

```
( <Test case name> | '*' )
```

Saves a test case as a system level Message Sequence Chart in the test cases directory (with file extension `.mpr`). The MSC contains a separate instance axis for each channel to the environment and another axis for the system. It comprises all signals sent to and from the environment for the trace from the root of the behavior tree to the current system

state. If '*' is given instead of a test case name, a generic test case name is generated.

Save-MSC-Test-Step

Parameters:

```
( <Test step name> | '*' )
```

Saves a test step as a system level Message Sequence Chart in the test steps directory (with file extension `.mpr`). The MSC contains a separate instance axis for each channel to the environment and another axis for the system. It comprises all signals sent to and from the environment for the trace from the root of the behavior tree to the current system state. If '*' is given instead of a test step name, a generic test step name is generated.

Save-Options

Parameters:

```
<File name>
```

Creates an Explorer command file with the name given as parameter. The file contains commands defining the options of the Explorer. If this file is loaded (using the command Include-File) the options will be restored to their saved values.

Save-Reports-as-MSC-Test-Cases

Parameters:

```
( <Report type> | '-' ) <Name prefix>
```

Saves all reports of a specified type as MSC test cases in the test cases directory (in system level MSC format; with file extension `.mpr`). The MSCs contain a separate instance axis for each channel to the environment and another axis for the system. For each MSC test case, a generic name is generated consisting of the name prefix, a four letter abbreviation of the report type and a five digit number. If '-' is provided instead of a report type, all current reports are saved.

Save-State-Space

Parameters:

```
<File name>
```

A Labelled Transition System (LTS) representing the generated state space is saved to a file. For a description of the file syntax, see section "State Space Files" on page 2384.

Note:

It is necessary to have executed an Exhaustive-Exploration command before the state space can be saved on a file.

Save-Test-Suite

Parameters:

```
<Test suite name> <File name>
```

Saves the generated test cases and all constraints in a test suite file in TTCN MP format. In addition, it generates the declarations part of the TTCN test suite. If the generation of a test case has been interrupted, Autolink prompts the user to decide whether to cancel the command or to abort the test case generation. The suggested file extension for the second parameter is `.mp`.

Note that this command requires a TTCN Link license in addition to the Explorer license.

Save-Test-Values

Parameters:

```
<File name>
```

A command file is generated containing Explorer commands that, if loaded with the Include-File command, will recreate the current test value definitions.

Scope

Parameters:

```
(None)
```

This command prints the current scope. See the command Set-Scope for a description of scope.

Scope-Down

Parameters:

```
<Optional service name>
```

Moves the scope one step down in the procedure call stack. If the current scope is a process containing services, one of the services should be specified. See also the commands Stack, Set-Scope and Scope-Up.

Scope-Up

Parameters:

```
(None)
```

Moves the scope one step up in the procedure call stack. Up from a service leads to the process containing the service. See also the commands Set-Scope, Stack and Scope-Down.

SDL-Trace

Parameters:

(None)

This command opens an SDL Editor that will show the transition leading to the current system state. Whenever the current system state changes in the Explorer, the SDL Editor will be updated.

If the SDL trace already is on, the command will turn it off.

Set-Application-All

Parameters:

(None)

The state space options of the Explorer are set to perform state space exploration according to the semantics of an application generated by the SDL Code Generator. No assumptions are made about the performance of the SDL system compared to timeout values or the performance of the environment.

This command sets the exploration mode factors by executing the following commands:

```
Define-Transition SDL
Define-Scheduling First
Define-Priorities 1 1 1 1 1
```

Set-Application-Internal

Parameters:

(None)

The state space options of the Explorer are set to perform state space exploration according to the semantics of an application generated by the SDL Code Generator. The assumption is made that the time it takes for the SDL system to perform internal actions is very small compared to timeout values and the response time of the environment.

This command sets the exploration mode factors by executing the following commands:

```
Define-Transition SDL
Define-Scheduling First
Define-Priorities 1 2 2 1 2
```

Set-Scope

Parameters:

```
<PId value> <Optional service name>
```

This command sets the scope to the specified process, at the bottom procedure call. If the process contains services, one of the services can be given as parameter to the command. A scope is a reference to a process instance, a reference to a service instance if the process contains services, and possibly a reference to a procedure instance called from this process/service. The scope is used for a number of other commands for examining the local properties of a process instance. The scope is automatically set to the process that executed in the transition leading to the current system state.

See also the commands Scope, Stack, Scope-Down and Scope-Up.

Set-Specification-All

Parameters:

```
(None)
```

The state space options of the Explorer are set to perform state space exploration according to the ITU semantics for SDL. No assumptions are made about the performance of the SDL system compared to timeout values, or the performance of the environment.

This command sets the exploration mode factors by executing the following commands:

```
Define-Transition Symbol-Sequence
Define-Scheduling All
Define-Priorities 1 1 1 1 1
```

Set-Specification-Internal

Parameters:

```
(None)
```

The state space options of the Explorer are set to perform state space exploration according to the ITU semantics for SDL. The assumption is made that the time it takes for the SDL system to perform internal actions is very small compared to timeout values and the response time of the environment.

This command sets the exploration mode factors by executing the following commands:

```
Define-Transition Symbol-Sequence
Define-Scheduling All
```

```
Define-Priorities 1 2 2 1 2
```

Show-Coverage-Viewer

Parameters:

(None)

This command starts the Coverage Viewer. The current test coverage is automatically loaded and a symbol coverage tree is presented. See chapter 47, *The SDL Coverage Viewer* for further description of the Coverage Viewer.

Show-Mode

Parameters:

(None)

This command displays a summary of the current execution mode and some other information about the current state of the SDL Explorer.

Show-Navigator

Parameters:

(None)

This command starts the Navigator tool to simplify interactive navigation in the behavior tree of the SDL system. See "The Navigator Tool" on page 2360 for more details.

Show-Options

Parameters:

(None)

The values of all options defined for the Explorer are printed, including the report action defined for each report type.

Show-Report-Viewer

Parameters:

(None)

This command starts the Report Viewer that presents a hierarchical view of all reports generated during state space explorations. See "The Report Viewer" on page 2363 for more details.

Show-Versions

Parameters:

(None)

The versions of the SDL to C Compiler and the runtime kernel that generated the currently executing program are presented.

Signal-Disable

Parameters:

```
<Signal> | '-'
```

Disables all test values defined for the given signal. '-' means *all signals*. Use Signal-Enable to start using them again. See also Channel-Disable. Test values for a signal are only used if both the signal and the channel that transports the signal are enabled. By default, all signals and channels are enabled.

Signal-Enable

Parameters:

```
<Signal> | '-'
```

Enables all test values defined for the given signal. '-' means *all signals*. See also Channel-Enable. Test values for a signal are only used if both the signal and the channel that transports the signal are enabled. By default, all signals and channels are enabled.

Signal-Reset

Parameters:

```
<Signal> | '-'
```

Removes all existing test values for the given signal, and defines a default set of test values instead, using the current test value settings for data types and signal parameters. '-' means *all signals*.

Stack

Parameters:

```
(None)
```

The procedure call stack for the PId/service defined by the scope is printed. For each entry in the stack, the type of instance (procedure/process/service), the instance name and the current state is printed. See also the commands Set-Scope, Scope-Down and Scope-Up.

Top

Parameters:

```
(None)
```

Go up in the behavior tree to the start of the current path (the root system state).

Translate-MSC-Into-Test-Case

Parameters:

```
( <File name> | '-' )
```

Translates an MSC into an internal test case data structure without performing a state space search. Additionally, constraints for all signals occurring in the test case are generated. At the end, all constraints with identical signal definitions are merged.

If the command parameter is specified as '-', all MSCs in the current test cases directory are transformed into test cases. Alternatively, several MSC test cases may be selected by a pattern enclosed in apostrophes. Such a pattern may include the special characters '*', '?', and '[...]' and is evaluated just as in a UNIX command shell.

Translate-MSC-Into-Test-Case does not require a fully specified SDL system. However, the signals used in the MSC must be defined in the SDL specification. Additionally, the channels between the SDL system and its environment have to be specified in order to find out which MSC instances relate to PCOs and which relate to the SDL system. Therefore, a direct translation cannot be applied to MSCs which consist of only a single instance axis for the environment.

Since the translation is done statically, it is not checked whether the MSC can in fact be verified. In addition, no inconclusive events are computed, unless they are stated explicitly by the use of an MSC exception expression.

Note that this command requires a TTCN Link license in addition to the Explorer license.

Tree-Search

Parameters:

```
(None)
```

Performs a tree search of the state space from the current system state. A tree search is an exploration where all possible combinations of actions are executed. The tree that is explored is exactly the same tree that can manually be inspected using the Navigator feature (or manual exploration using the Next / List-Next commands).

The depth of the tree search is bounded and is defined by the Define-Tree-Search-Depth command. The default depth is 100.

The command can be aborted by the user by pressing the *Break* button in the Explorer UI or any key if running the explorer from the command prompt.

Tree-Walk

Parameters:

```
<Timeout> <Coverage>
```

Performs an automatic exploration of the state space starting from the current system state. In contrast to Random-Walk, Tree-Walk is based on a deterministic algorithm that performs a sequence of tree searches with increasing depth starting at various states in the reachability graph. Tree Walk combines the advantages of both the depth-first and breadth-first search strategy - it is able to visit states located deep in the reachability graph and to find a short path to a particular state at the same time. Tree Walk is guided by a symbol coverage heuristic. Therefore it is particularly suitable for automatic test case generation.

Computation stops if either time exceeds <Timeout> (specified in minutes) or the targeted coverage (specified in percent) is reached. Alternatively, the exploration can be stopped at any time by pressing *Return* at the command prompt or by pressing the *Break* button if the graphical user interface is used.

Tree Walk creates a number of “TreeWalk” reports. These reports can be used for the generation of MSC test cases by applying command Save-Reports-as-MSC-Test-Cases.

Up

Parameters:

```
<Number of levels>
```

Go up the specified number of levels in the behavior tree. Up 1 goes to the parent state of the current system state. Up will stop at the root of the behavior tree (the start state) if the parameter is too large.

Verify-MSC

Parameters:

```
<MSC file>
```

This command will perform an automatic exploration from the current system state to search for all execution traces that are consistent with the MSC that is given as a parameter. The search algorithm is a variant of the bit state algorithm that is adapted to suite MSC verification. Before

the exploration is started, the root of the behavior tree is set to the current system state.

The file can either be an basic MSC file (`.msc`), a textual MSC file (`.mpr`) or a high-level MSC file (`.mrm`).

When the exploration is finished, the same information as for a bit state exploration is printed. In addition, a message is printed stating whether or not the MSC was verified, i.e. if an execution path was found that satisfied the MSC, and reports for MSC verification and MSC violation are generated.

If the MSC is a high-level MSC or it contains MSC reference expressions more that one MSC verification report can be generated. There will be one report for each sequence of 'leaf' MSCs that can be verified. For more information on criteria for MSC verification, see "MSC Verification Errors" on page 2373.

If the MSC was verified and exactly one MSC verification report was generated, the system state of the Explorer will be set up to the last system state in the path that satisfied the MSC. Otherwise, the system is returned to its initial state.

If the MSC contains MSC references the Explorer needs to match the logical MSC names in the MSC references with MSC file names. This is done according to the following algorithm:

1. If the Autolink test step directory is set, then the MSC file is assumed to be located in this directory and have the same name as the MSC with the suffix `.msc`, `.mpr` or `.mrm`.
2. The Explorer then checks if there is an MSC with the correct name in the same module in the Organizer as the parent MSC. If this is the case this MSC file is used.
3. If none of above applies then the Explorer checks if there is an MSC file in the same directory as the parent MSC with the correct name and with the suffix `.msc`, `.mpr` or `.mrm`. If this is the case this MSC file is used.

Graphical User Interface

This section describes the appearance and functionality of the graphical user interface to the explorer monitor (ExpUI). Some user interface descriptions general to all tools can be found in chapter 1, *User Interface*

and Basic Operations. These general descriptions are not repeated in this chapter.

> **Note:**
>
> The ExpUI looks and behaves very much like the SimUI, the graphical user interface to the simulator monitor. The differences are the set of available monitor commands and button modules, and a few additional menus and sub windows.

For an explanation of the Explorer main window, depicted in Figure 484, see "Graphical User Interface" on page 2199 in chapter 49, *The SDL Simulator*.

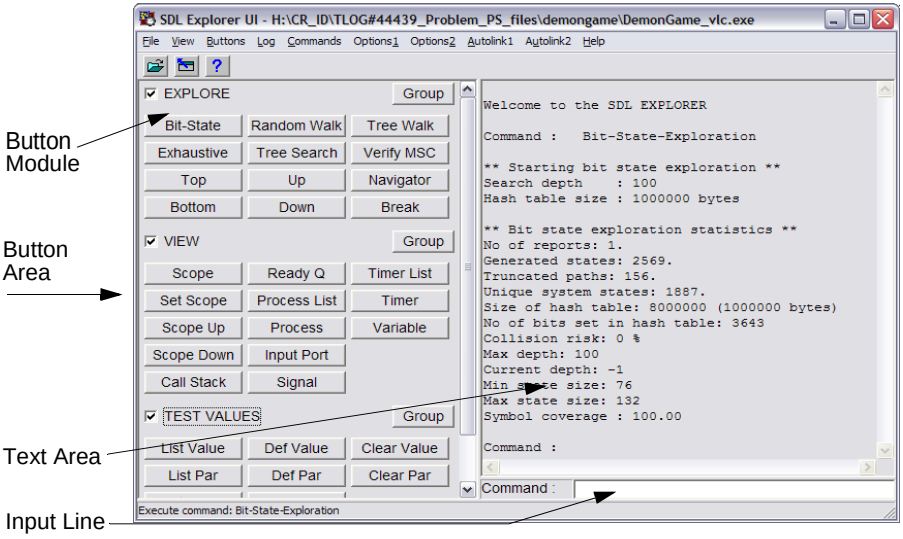


Figure 484: The main window

Starting the ExpUI

A new ExpUI is started by selecting *SDL > Explorer UI* from the *Tools* menu in the Organizer. When the ExpUI is started, a number of definition files are read, controlling the contents of the main window and some status windows. See "Definition Files" on page 2365 for more information.

No explorer is started automatically by the ExpUI in this way. The user must start an explorer by selecting *Open* from the *File* menu, as stated in the text area of the main window, or by using the *Open* quick button.

A simple way to generate an explorer, start the ExpUI and open the explorer is to click the *Explore* quick button in the Organizer.

When a simulator is started, a file selection dialog may be opened if the SDL system contains external synonyms. For more information, see "Supplying Values of External Synonyms" on page 2396 in chapter 53, *Validating a System*.

The Default Button Modules

The following tables list the default buttons in the button modules and the corresponding monitor command. See "Monitor Commands" on page 2300 for more information.

> **Note:**
>
> The buttons in the button modules are specified in the button definition file. If the default button file is not used, the button modules may be different than described here. See "Button and Menu Definitions" on page 2222 in chapter 49, *The SDL Simulator* for more information.

The *EXPLORE* Module

Button	Monitor command
Bit-State	Bit-State-Exploration
Exhaustive	Exhaustive-Exploration
Top	Top
Bottom	Bottom
Random Walk	Random-Walk
Tree Search	Tree-Search
Up	Up 1
Down	Down 1
Tree Walk	Tree-Walk
Verify MSC	Verify-MSC
Navigator	Show-Navigator
Break	Pressing `<Return>`

The *VIEW* Module

Button	Monitor command
Scope	Scope
Set Scope	Set-Scope
Scope Up	Scope-Up
Scope Down	Scope-Down
Call Stack	Stack
Ready Q	List-Ready-Queue
Process List	List-Process
Process	Examine-PId
Input Port	List-Input-Port
Signal	Examine-Signal-Instance

Button	Monitor command
Timer List	List-Timer
Timer	Examine-Timer-Instance
Variable	Examine-Variable

The *TEST VALUES* Module

Button	Monitor command
List Value	List-Test-Values
List Par	List-Parameter-Test-Values
List Signal	List-Signal-Definitions
Clear Signal	Clear-Signal-Definitions
Def Value	Define-Test-Value
Def Par	Define-Parameter-Test-Value
Def Signal	Define-Signal
Extract Sigs	Extract-Signal-Definitions-From-MSC
Clear Value	Clear-Test-Values
Clear Par	Clear-Parameter-Test-Values

The Menu Bar

This section describes the additional menus of the ExpUI's menu bar in comparison with the SimUI. The menus common to both the ExpUI and the SimUI is described in "The Menu Bar" on page 2207 in chapter 49, *The SDL Simulator*.

The menu bar contains the following menus:

- *File Menu*
  (See "File Menu" on page 2208 in chapter 49, *The SDL Simulator*.)
- *View Menu*
  (See "View Menu" on page 2208 in chapter 49, *The SDL Simulator*.)
- *Buttons Menu*
  (See "Buttons Menu" on page 2209 in chapter 49, *The SDL Simulator*.)
- *Log Menu*
  (See "Log Menu" on page 2210 in chapter 49, *The SDL Simulator*.)
- *Help Menu*
  (See "Help Menu" on page 15 in chapter 1, *User Interface and Basic Operations*.)
- *Commands Menu*
- *Options1 Menu*
- *Options2 Menu*
- Autolink1 Menu
- Autolink2 Menu

Additional SDL Explorer Menus

In addition to the standard SimUI menus, a few special explorer menus are included in the menu bar. The menu choices in these menus simply execute a monitor command, i.e. they are functionally equivalent to buttons in the button modules. If the monitor command requires parameters, they are prompted for using dialogs in the same way as the command buttons.

The following tables list the default menu choices and the corresponding monitor command. See "Monitor Commands" on page 2300 for more information.

> **Note:**
>
> The additional menus in the ExpUI are specified in the button definition file. If the default button file is not used, the additional menus may be different than described here. See "Button and Menu Definitions" on page 2222 in chapter 49, *The SDL Simulator* for more information.

Commands Menu

Menu choice	Monitor command
Toggle MSC Trace	MSC-Trace
Toggle SDL Trace	SDL-Trace
Show Report Viewer	Show-Report-Viewer
Show Coverage Viewer	Show-Coverage-Viewer
Show Navigator	Show-Navigator
Define Rule	Clear-Rule ; Define-Rule
Include Command Script	Include-File

Options1 Menu

Menu choice	Monitor command
Show Options	Show-Options
Reset	Reset
Default	Default-Options
Advanced	Define-Scheduling All ; Define-Priorities 1 1 1 1 1 ; Define-Max-Input-Port-Length 2 ; Define-Report-Log MaxQueuelength Off
State Space : Transition	Define-Transition
- : Scheduling	Define-Scheduling
- : Priorities	Define-Priorities
- : Input Port Length	Define-Max-Input-Port-Length
- : Transition Length	Define-Max-Transition-Length
- : Max Instance	Define-Max-Instance
- : Timer Progress	Define-Timer-Progress
- : Channel Queues	Define-Channel-Queue
- : Max State Size	Define-Max-State-Size
- : Symbol Time	Define-Symbol-Time
Report : Continue	Define-Report-Continue
- : Prune	Define-Report-Prune
- : Abort	Define-Report-Abort
- : Report Log	Define-Report-Log
MSC Trace Auto Popup	Define-MSC-Trace-Autopopup
Report Viewer Auto Popup	Define-Report-Viewer-Autopopup

Options2 Menu

Menu choice	Monitor command
Show Options	Show-Options
Bit-State : Hash Size	Define-Bit-State-Hash-Table-Size
- : Depth	Define-Bit-State-Depth
- : Iteration Step	Define-Bit-State-Iteration-Step
Random Walk : Repetitions	Define-Random-Walk-Repetitions
- : Depth	Define-Random-Walk-Depth
Exhaustive : Depth	Define-Exhaustive-Depth
Tree Search : Depth	Define-Tree-Search-Depth
MSC Ver : Timer Check Level	Define-Timer-Check-Level
- : Condition Check	Define-Condition-Check
- : Depth	Define-MSC-Verification-Depth
Autolink : Concurrent TTCN	Define-Concurrent-TTCN
- : Depth	Define-Autolink-Depth
- : Generation Mode	Define-Autolink-Generation-Mode
- : Global Timer	Define-Global-Timer
- : Hash Table Size	Define-Autolink-Hash-Table-Size
- : Options	Define-Autolink-State-Space-Options
- : Signal Mapping	Define-TTCN-Signal-Mapping
- : Test Cases Directory	Define-MSC-Test-Cases-Directory
- : Test Steps Directory	Define-MSC-Test-Steps-Directory
- : Test Steps Format	Define-TTCN-Test-Steps-Format

Autolink1 Menu

Menu choice	Monitor command
MSC : Save Test Case	Save-MSC-Test-Case

Menu choice	Monitor command
- : Save Reports	Save-Reports-as-MSC-Test-Cases
- : Save Error Reports	Save-Error-Reports-As-MSCs
- : Save Test Step	Save-MSC-Test-Step
- : List	List-MSC-Test-Cases-And-Test-Steps
- : Clear Test Case	Clear-MSC-Test-Case
- : Clear Test Step	Clear-MSC-Test-Step
Test case : Generate	Generate-Test-Case
- : Translate	Translate-MSC-Into-Test-Case
- : List	List-Generated-Test-Cases
- : Print	Print-Generated-Test-Case
- : Clear	Clear-Generated-Test-Case
- : Save	Save-Generated-Test-Case
- : Load	Load-Generated-Test-Cases
Test suite : Save	Save-Test-Suite

Autolink2 Menu

Menu choice	Monitor command
Configuration : Define	Define-Autolink-Configuration
- : Print	Print-Autolink-Configuration
- : Clear	Clear-Autolink-Configuration
- : Save	Save-Autolink-Configuration
- : Load	Include-File
Constraint : Define	Define-Constraint
- : List	List-Constraints
- : Rename	Rename-Constraint
- : Merge	Merge-Constraints
- : Parameterize	Parameterize-Constraint
- : Clear	Clear-Constraint
- : Save	Save-Constraint
- : Load	Load-Constraints
Timer : Define	Define-Timer-Declaration

The Command and Watch Windows

The Command window of the ExpUI is identical to the SimUI (see "Command Window" on page 2216 in chapter 49, *The SDL Simulator*). The only difference is that the default commands to execute are "List-Process -" and "Print-Trace 1". The set of commands to execute are stored in a *command definition file* (see "Definition Files" on page 2365). The default command definition file can be changed with the Preference Manager.

The Watch window of the ExpUI is identical to the SimUI (see "Watch Window" on page 2219 in chapter 49, *The SDL Simulator*).

The Navigator Tool

The Navigator is a separate tool in the ExpUI that is used for navigating in the behavior tree. It can be opened in the following ways:

- By issuing the monitor command Show-Navigator
- By selecting *Show Navigator* from the *Commands* menu.
- By clicking the *Navigator* button in the button area.

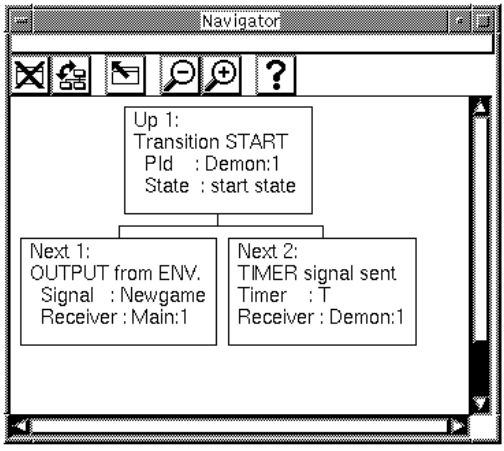


Figure 485: The Navigator window

The Navigator window consists of the tree area and the tool bar. There is no menu bar or status bar in the Navigator window.

Tree Area

The tree area of the Navigator window shows the structure of the behavior tree around the current system state. Each box, or node, represents a tree transition from one system state to another.

- The up node, always labelled *Up 1*, shows the tree transition leading to the current system state.
- The down nodes, labelled *Next 1*, *Next 2*, etc., show the possible tree transitions from the current system state. One of the down nodes may be labelled with three asterisks "***" to show which node is part of the current path, i.e., which part of the tree that already has been explored.

The text in the boxes contains the textual trace describing the tree transition. This may represent a complete or partial SDL process graph transition, depending on how the behavior tree is set up. The number of down nodes also depends on the structure of the behavior tree. This is determined by the state space options; see "State Space Options" on page 2466 in chapter 53, *Validating a System*.

Double-clicking a node in the Navigator executes the corresponding tree transition and moves one level up or down in the behavior tree. The current system state is changed and the Navigator window is updated to show the situation around the new system state.

> **Note:**
> Double-clicking a collapsed node (see below) **does not** expand the node; it always executes the corresponding transition.

The Navigator window is also updated whenever a monitor command is executed that changes the current system state.

Popup Menus

Each node in the node area has an associated popup menu.

Up 1	**On the up node:** Go up one level in the tree. This is the same as double-clicking the up node.
Up to top	**On the up node:** Go to the top of the behavior tree.

Continue Up	**On the up nodes**: Go up in the tree until more than one transition is possible (see Continue-Up-Until-Branch command).
Goto	**On the down nodes:** Go down this branch of the tree. This is the same as double-clicking the down node.
Continue	**On the down nodes:** Go down this branch of the tree until more than one transition is possible (see Continue-Until-Branch command).
Expand	Expand the collapsed node to show the down nodes. (Not applicable on a down node.)
Expand Substructure	The same as *Expand*.
Collapse	Collapse the node to hide the down nodes. A small triangle below the node shows that it is collapsed. (Not applicable on a down node.)

Quick Buttons

The following quick buttons are special to the Navigator tool. The general quick buttons are described in "General Quick-Buttons" on page 24 in chapter 1, *User Interface and Basic Operations*.

Close
Closes the Navigator tool.

Structure
Switches between a tree structure and a list structure in the node area.

Show Explorer UI
Raises the Explorer UI window.

The Report Viewer

The Report Viewer is a separate tool in the ExpUI that is used for examining the reports generated during an automatic state space exploration. It is opened in one of the following ways:

- Automatically when an automatic state space exploration has finished, unless the option Report Viewer Autopopup is off
- By issuing the monitor command Show-Report-Viewer
- By selecting *Show Report Viewer* from the *Commands* menu.

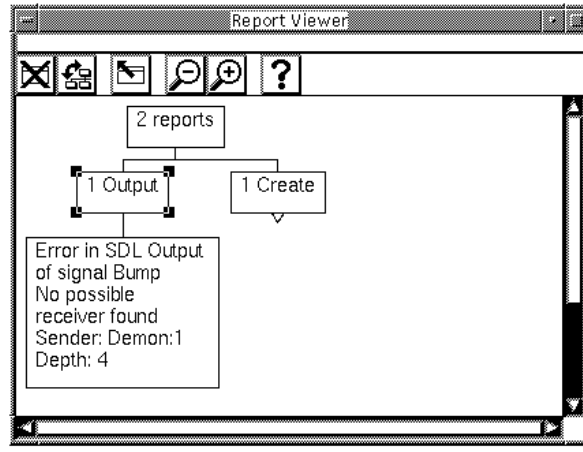


Figure 486: The Report Viewer

The Report Viewer window consists of the Report Area, containing Popup Menus, and the tool bar, containing Quick Buttons. There is no menu bar or status bar in the Report Viewer window.

Report Area

The report area of the Report Viewer shows the current reports from the latest state space exploration in the form of a report tree. The report tree contains three levels of nodes (see Figure 486):

- The top node shows how many reports there are. This node is always visible.

- The next level contains one node for each *report type*. The text shows the report type and how many reports of that type there are. When the Report Viewer is started, the report type nodes are shown, but they are collapsed (indicated by a small triangle below the node). For a list of possible report types, see "Rules Checked During Exploration" on page 2366.
- The bottom level contains one node for each actual report. The text shows the error or warning message, and the depth in the behavior tree where the report was generated. If the same report has been generated more than once, only one report is shown; the one generated at the lowest depth in the tree. When the Report Viewer is started, the report nodes are not visible.

Double-clicking a report node at the bottom level causes the Explorer to "go to" the report, i.e., go to the system state where the report was generated. The following things happen:

- The textual trace from the transition where the report was generated is printed in the text area of the main window.
- If the Navigator tool is opened, it is updated to show the transition where the report was generated and the error message.
- An MSC Editor is opened, showing the MSC trace from the start state to the state where the report was generated. The MSC Editor is not opened if the option MSC Trace Autopopup is off.

Double-clicking the top node or a report type node will expand or collapse the node, depending on its state. A collapsed node hides its underlying nodes and is indicated by a small triangle below the node.

Popup Menus

Goto	**Only on report nodes:** Go to the report. This is the same as double-clicking the report node.
Expand	Expand the collapsed node to show the nodes in the next level of the tree. This is the same as double-clicking the collapsed node (top node or report type node). (Not applicable on a report node.)

Expand Substructure	Expand the collapsed node to show all nodes on all levels below it. (Not applicable on a report node.)
Collapse	Collapse the node to hide all nodes on all levels below it. This is the same as double-clicking the node (top node or report type node). (Not applicable on a report node.)

Quick Buttons

The following quick buttons are special to the Report Viewer. The general quick buttons are described in "General Quick-Buttons" on page 24 in chapter 1, *User Interface and Basic Operations*.

Close
Closes the Report Viewer.

Structure
Switches between a tree structure and a list structure in the report area.

Show Explorer UI
Raises the Explorer UI window.

Definition Files

In the ExpUI, the syntax and contents of the definition files are the same as for the SimUI; see "Definition Files" on page 2221 in chapter 49, *The SDL Simulator*. The default file names for the definition files are `val_def.btns`, `val_def.cmds` and `val_def.vars`, respectively.

Rules Checked During Exploration

For each system state encountered during state space exploration, a number of rules are checked to detect errors or possible problems in the SDL system. If a rule is satisfied, a report is generated to the user.

The rules that are checked by the SDL Explorer are the same ones as checked and reported by the SDL Simulator, with a few additions and differences.

Apart from the predefined rules, an additional rule can be defined by the user of the explorer to check for other properties of the system states encountered. See "User-Defined Rules" on page 2376 for more information.

The rules are listed below, together with the reported error messages. The rules are grouped according to the corresponding report type, used in the Report Viewer in the Explorer. The names of the report types listed below are the ones to be used in the monitor commands that define the report actions.

Deadlock

Report Type: Deadlock

```
Deadlock
```

All processes instances are waiting for some other process instance to act, implying that none of the processes will execute. This is referred to as a deadlock.

Implicit Signal Consumption

Report Type: ImplSigCons

```
Warning: Implicit signal consumption of signal
<signal>
```

A signal was sent to a process that was not able to handle (or save) the signal in the current state, so it was implicitly consumed.

Create Errors

Report Type: Create

```
Error in SDL Create: Processtype <type>
More static instances than maximum number of
instances.
```

There are more static instances defined than the maximum allowed number of instances.

```
Warning in SDL Create: Processtype <type>
Unsuccessful create. Number of instances has reached
maximum number.
```

An attempt has been made to create a new instance of a process type of which there is already the maximum number of allowed instances active. The maximum number is either the value defined by the command Define-Max-Instance, or the value defined in the SDL diagram, whichever is smallest.

Output and Remote Procedure Call Errors

Report Type: Output

```
Error in SDL Output of signal <signal>
No valid receiver found
```

An attempt was made to output a signal to an invalid PId expression.

```
Error in SDL Output of signal <signal>
No path to receiver
```

An attempt was made to output a signal to a PId expression. There exists, however, no path of channels and signal routes between the sender and the receiver that can convey the signal.

```
Error in SDL Output of signal <signal>
No possible receiver found
```

An attempt was made to output a signal without specifying a PId expression. When all paths or all paths mentioned in a via clause had been examined no possible receiver was found.

```
Error in SDL Output of signal <signal>
Several possible receivers found
```

An attempt was made to output a signal without specifying a PId expression. When all paths or all paths mentioned in a via clause had been examined several possible receivers were found.

```
Error in SDL Output of signal <signal>
Signal sent to stopped process instance
```

An attempt was made to output a signal to a PId expression that referred to a process instance which has performed a stop action.

```
Error in SDL Output of signal <signal>
Signal sent to NULL
```

An attempt was made to output a signal to a PId expression that was null.

```
Error in SDL Output of signal <signal>
Illegal signal type in output TRANSFER
```

An attempt was made to output a signal with the TRANSFER directive that was not the same signal as in the preceding input.

```
Error in remote procedure call: <name>
More than one process provides the remote procedure
```

An attempt was made to call a remote procedure in a system state where there are more than one possible process instance that can provide the remote procedure.

```
Error in remote procedure call: <name>
No process provides the remote procedure
```

An attempt was made to call a remote procedure in a system state where there is no possible process instance that can provide the remote procedure.

Max Queue Length Exceeded

Report Type: MaxQueuelength

```
Error in SDL Output of signal <signal>
Max input port length exceeded
```

The length of the input port of the receiving process has exceeded the value defined by the monitor command Define-Max-Input-Port-Length.

```
Error in SDL Output of signal <signal>
Max channel queue length exceeded
```

The length of the queue of the receiving channel has exceeded the value defined by the monitor command Define-Max-Input-Port-Length.

```
Error in channel output. Max input port length
exceeded
```

The length of the input port of the receiving process has exceeded the value defined by the monitor command Define-Max-Input-Port-Length.

```
Error in channel output. Max channel queue length
exceeded
```

The length of the queue of the receiving channel has exceeded the value defined by the monitor command Define-Max-Input-Port-Length

Channel Output Errors

Report Type: ChannelOutput

In addition to the following messages, information about the channel, signal, sender and receiver is also displayed.

```
Error in channel output. No valid receiver found
```

An attempt was made to output a signal to an invalid PId expression.

```
Error in channel output. No path to receiver
```

An attempt was made to output a signal to a PId expression. There exists, however, no path of channels and signal routes between the sender and the receiver that can convey the signal.

```
Error in channel output. No possible receiver found
```

An attempt was made to output a signal without specifying a PId expression. When all paths or all paths mentioned in a via clause had been examined no possible receiver was found.

```
Error in channel output. Several possible receivers
found
```

An attempt was made to output a signal without specifying a PId expression. When all paths or all paths mentioned in a via clause had been examined several possible receivers were found.

```
Error in channel output. Signal sent to stopped
process instance
```

An attempt was made to output a signal to a PId expression that referred to a process instance which has performed a stop action.

```
Error in channel output. Signal sent to NULL
```

An attempt was made to output a signal to a PId expression that was null.

Operator Errors

Report Type: Operator

The errors that can be found in operators defined in the predefined data types are listed below.

```
Error in SDL Operator: Modify! in sort Charstring,
Index out of bounds

Error in SDL Operator: Extract! in sort Charstring,
Index out of bounds

Error in SDL Operator: MkString in sort Charstring,
character NUL not allowed

Error in SDL Operator: First in sort Charstring,
Charstring length is zero

Error in SDL Operator: Last in sort Charstring,
Charstring length is zero

Error in SDL Operator: Substring in sort Charstring,
Charstring length is 0

Error in SDL Operator: Substring in sort Charstring,
Sublength is less than or equal to zero

Error in SDL Operator: Substring in sort Charstring,
Start is less than or equal to zero

Error in SDL Operator: Substring in sort Charstring,
Start + Substring length is greater than string
length.

Error in SDL Operator: Division in sort Integer,
Integer division by 0.

Error in SDL Operator: Division in sort Real, Real
division by 0.

Error in SDL Operator: Fix in sort Integer
Integer overflow
Second operand is 0

Error in SDL Operator: Mod in sort Integer
Second operand is 0

Error in SDL Operator: Rem in sort Integer
Second operand is 0
```

Range Errors

Report Type: Subrange

```
Error in assignment in sort <sort>:
<value> out of range
```

A variable of a restricted syntype is assigned a value out of its range.

Index Error

Report Type: Index

```
Error in SDL array index in sort <sort>:
<value> out of range
```

An array index is out of range.

Decision Error

Report Type: Decision

```
Error in SDL Decision: Value is <value>:
Fatal error. Analysis is not continued below this
node
```

The value of the expression in the SDL decision did not match any of the possibilities (answers).

Import Errors

Report Type: Import

Error during execution of an import statement. Supplementary information about remote variables and exporting processes is also given.

```
Error in SDL Import. Attempt to import from the
environment

Error in SDL Import. Attempt to import from NULL

Error in SDL Import. Attempt to import from stopped
process instance

Error in SDL Import. Several processes exporting
this variable

Error in SDL Import. The specified process does not
export this variable

Error in SDL Import. No process exports this
variable

Error in SDL Import. More than one process exports
this variable
```

View Errors

Report Type: View

Error during execution of view statement. Supplementary information about viewed variables and revealing processes is also given.

```
Error in SDL View. Attempt to view from the
environment

Error in SDL View. Attempt to view from NULL

Error in SDL View. Attempt to view from stopped
process instance

Error in SDL View. More than one process instance
reveals the variable

Error in SDL View. The specified process does not
reveal the variable.

Error in SDL View. No process instance reveals this
variable
```

Transition Length Error

Report Type: MaxTransLen

```
Max transition length exceeded
```

The maximum number of SDL symbols executed in one behavior tree transition is more than the value defined by the monitor command Define-Max-Transition-Length.

Non Progress Loop Error

Report Type: Loop

```
Loop Detected.
```

The Explorer includes a mechanism for non-progress loop detection. A report will be generated if the Explorer during state space exploration finds a loop in the state space which does not contain any progress transition. A progress transition is defined as a transition that either:

- contains communication with the environment
- contains an assignment of the variable `xProgress` to 1 in user-defined C code
- includes a timer expiration (see "Timer Progress" on page 2470 in chapter 53, *Validating a System*)
- includes a spontaneous transition (see "Spontaneous Transition Progress" on page 2471 in chapter 53, *Validating a System*)

Assertion Errors

Report Type: Assertion

```
Assertion is false: <user-defined string>
```

A user-defined action in the SDL system has called the function `xAssertError`. See "Using Assertions" on page 2457 in chapter 53, *Validating a System* for more information.

User Defined Rule

Report Type: UserRule

```
User-defined rule satisfied
```

A user-defined rule is evaluated to true.

Observer Errors

Report Type: Observer

```
Observer violation
```

A process defined as an observer process has not been able to execute a transition.

MSC Verification Errors

Report Type: MSCVerification
Report Type: MSCViolation

```
MSC <MSC name> verified
MSC sequence: <MSC name list>

MSC <MSC name> violated
Event: <SDL Event>
```

The MSC verification report is given when a state space exploration has reached a state where the execution trace from the root of the behavior tree to the current state satisfies the loaded MSC. In this case, "satisfies" means that:

- The execution trace must include all events that exist in the MSC.
- The execution trace must not contain any observable event that is not part of the MSC.
- The sequence of observable events in the execution trace must be consistent with the partial ordering of the events that is defined by the MSC.

An event on the execution trace is considered to be “observable” if it is either

- an input or output of a signal that is sent between process instances that correspond to different MSC instances, or
- a create of a process instance that corresponds to a different MSC instance than the creating process instance, or
- a set, reset or input of a timer, depending on the value set by the command Define-Timer-Check-Level.

Note:

Two process instances can correspond to the same MSC instance if the MSC is a system or block level MSC.

The MSC violation report is given during state space exploration for each generated execution trace that either:

- includes an observable event that is not part of the loaded MSC, or
- contains a sequence of observable events that is not consistent with the partial ordering of events defined by the MSC.

REF Errors

Report Type: RefError

```
No reference to dynamically allocated memory of
sort: <Sort Name>
```

The list of dynamically allocated data areas that is maintained by the Explorer contains a data area that no SDL entity (like a variable, signal parameter etc.) references. This indicates a memory leak in the system.

```
No reference to dynamically allocated memory
```

The list of dynamically allocated data areas that is maintained by the Explorer contains a data area that no SDL entity (like a variable, signal parameter etc.) references. This indicates a memory leak in the system. The lack of <Sort Name> in this report indicates that this is a newly allocated data area and there has been no previous system state with a reference to this data area.

```
Referenced data area not found:
Variable <Variable name> in process <Process name>
```

A REF variable (or signal parameter, structure field etc.) has been found that references a data area that is not allocated. To avoid this report always set the REF variables to NULL after the data area has been released.

```
Dereferencing a NULL pointer of sort: <Sort Name>.
```

An expression `XX*>` has been evaluated where `XX` was NULL.

```
Calling UserMalloc with parameter NULL
Calling UserFree with parameter NULL
```

The `UserMalloc` or `UserFree` functions has been called with NULL as a parameter which is illegal.

```
Calling UserFree without dynamically allocated
memory
```

`UserFree` has been called with a parameter that is not a pointer to one of the data areas in the list of the dynamically allocated data areas.

User-Defined Rules

User-defined rules are used during state space exploration to check for properties of the system states encountered. If a system state is found for which a user-defined rule is true, this will be listed among the other reports when giving the List-Reports command. During an exploration more that one user defined rule report can be generated. There will be one report for each value assignment that can be made to a rule. The value assignments are the values printed by the Print-Evaluated-Rule command.

A rule essentially gives the possibility to define predicates that describe properties of one particular system state. A rule consists of a predicate (as described below) followed by a semicolon (';'). In a rule, all identifiers and reserved words can be abbreviated as long as they are unique.

Note:

Only one rule can be used at any moment. If more than one rule is needed, reformulate the rules as one rule, using the boolean operators described below.

Predicates

The following types of predicates exist:

- Quantifiers over process instances and signals in input ports.
- Boolean operator predicates such as "and", "not" and "or".
- Relational operator predicates such as "=" and ">".

Parenthesis are allowed to group predicates.

Quantifiers

The quantifiers listed below are used to define rule variables denoting process instances or signals. The rule variables can be used in process or signal functions described later in this section.

```
exists <RULE VARIABLE> [: <PROCESS TYPE>]
[ | <PREDICATE>]
```

This predicate is true if there exists a process instance (of the specified type) for which the specified predicate is true. Both the process type and the predicate can be excluded. If the process type is excluded all process

instances are checked. If the predicate is excluded it is considered to be true.

```
all <RULE VARIABLE> [ : <PROCESS TYPE>]
[ | <PREDICATE>]
```

This predicate is true for all process instances (of the specified type) for which the specified predicate is true. Both the process type and the predicate can be excluded. If the process type is excluded all process instances are checked. If the predicate is excluded it is considered to be true.

```
siexists <RULE VARIABLE> [ : <SIGNAL TYPE>]
[ -  <PROCESS INSTANCE>] [ |  <PREDICATE>]
```

This predicate is true if there exists a signal (of the specified type) in the input port of the specified process for which the specified predicate is true. If no signal type is specified, all signals are considered. If no process instance is specified the input ports of all process instances are considered. If no predicate is specified it is considered to be true. The specified process can be either a rule variable that has previously been defined in an `exists` or `all` predicate, or a process instance identifier (`<PROCESS TYPE>:<INSTANCE NO>`).

```
siall <RULE VARIABLE> [ :  <SIGNAL TYPE>]
[ - <PROCESS INSTANCE>] [ | <PREDICATE>]
```

This predicate is true for all signals (of the specified type) in the input port of the specified process for which the specified predicate is true. If no signal type is specified all signals are considered. If no process is specified the input ports of all process instances are considered. If no predicate is specified it is considered to be true. The specified process can be either a rule variable that has previously been defined in an `exists` or `all` predicate, or a process instance identifier (`<PROCESS TYPE>:<INSTANCE NO>`).

Boolean Operator Predicates

The following boolean operators are included (with the conventional interpretation):

```
not <PREDICATE>
<PREDICATE> and <PREDICATE>
<PREDICATE> or <PREDICATE>
```

The operators are listed in priority order, but the priority can be changed by using parenthesis.

Relational Operator Predicates

The following relational operator predicates exist:

```
<EXPRESSION> = <EXPRESSION>
<EXPRESSION> != <EXPRESSION>
<EXPRESSION> < <EXPRESSION>
<EXPRESSION> > <EXPRESSION>
<EXPRESSION> <= <EXPRESSION>
<EXPRESSION> >= <EXPRESSION>
```

The interpretation of these predicates is conventional. The operators are only applicable to data types for which they are defined.

Expressions

The expressions that are possible to use in relational operator predicates are of the following categories:

- Process functions: Extract values from process instances.
- Signal functions: Extract values from signals.
- Global functions: Examine global aspects of the system state.
- SDL literals: Conventional SDL constant values.

Process Functions

Most of the process functions must have a process instance as a parameter. This process instance can be either a rule variable that has previously been defined in an `exists` or `all` predicate, a process instance identifier (`<PROCESS TYPE>:<INSTANCE NO>`), or a function that returns a process instance, e.g. `sender` or `from`.

```
state( <PROCESS INSTANCE> )
```

Returns the current SDL state of the process instance.

```
type( <PROCESS INSTANCE> )
```

Returns the type of the process instance.

```
iplen( <PROCESS INSTANCE> )
```

Returns the length of the input port queue of the process instance.

```
sender( <PROCESS INSTANCE> )
```

Returns the value of the imperative operator sender (a process instance) for the process instance.

```
parent( <PROCESS INSTANCE> )
```

Returns the value of the imperative operator parent (a process instance) for the process instance.

```
offspring( <PROCESS INSTANCE> )
```

Returns the value of the imperative operator offspring (a process instance) for the process instance.

```
self( <PROCESS INSTANCE> )
```

Returns the value of the imperative operator self (a process instance) for the process instance.

```
signal( <PROCESS INSTANCE> )
```

Returns the signal that is to be consumed if the process instance is in an SDL state. Otherwise, if the process instance is in the middle of an SDL process graph transition, it returns the signal that was consumed in the last input statement.

```
<PROCESS INSTANCE> -> <VARIABLE NAME>
```

Returns the value of the specified variable. If `<PROCESS INSTANCE>` is a previously defined rule variable, the `exists` or `all` predicate that defined the rule variable must also include a process type specification.

```
<RULE VARIABLE>
```

Returns the process instance value of `<RULE VARIABLE>`, which must be a rule variable bound to a process instance in an `exists` or `all` predicate.

Signal Functions

Most of the signal functions must have a signal as a parameter. This signal can be either a rule variable that has previously been defined in an `siexists` or `siall` predicate, or a function that returns a signal, e.g. `signal`.

```
sitype( <SIGNAL> )
```

Returns the type of the signal.

```
to( <SIGNAL> )
```

Returns the process instance value of the receiver of the signal.

```
from( <SIGNAL> )
```

Gives the process instance value of the sender of the signal.

`<RULE VARIABLE> -> <PARAMETER NUMBER>`

Returns the value of the specified signal parameter. The `siexists` or `siall` predicate that defined the rule variable must also include a signal type specification.

`<RULE VARIABLE>`

Returns the signal value of `<RULE VARIABLE>`, which must be a rule variable bound to a signal in a `siexists` or `siall` predicate.

Global Functions

`maxlen( )`

Gives the length of the longest input port queue in the system.

`instno( [<PROCESS TYPE>] )`

Returns the number of instances of type `<PROCESS TYPE>`. If `<PROCESS TYPE>` is excluded the total number of process instances is returned.

`depth( )`

Gives the depth of the current system state in the behavior tree/state space.

SDL Literals

`<STATE ID>`

The name of an SDL state.

`<PROCESS TYPE>`

The name of a process type.

`<PROCESS INSTANCE>`

A process instance identifier of the format `<PROCESS TYPE>:<INSTANCE NO>`, e.g. Initiator:1.

`<SIGNAL TYPE>`

The name of a signal type.

`null`

SDL null process instance value

```
env
```

Returns the value of the process instance in the environment that is the sender of all signals sent from the environment of the SDL system.

```
<INTEGER LITERAL>
true
false
<REAL LITERAL>
<CHARACTER LITERAL>
<CHARSTRING LITERAL>
```

Autolink Configuration Syntax

An Autolink configuration is created with the command Define-Autolink-Configuration. The syntax of an Autolink configuration is expressed below in EBNF format.

```
<Start>          ::= "Define-Autolink-Configuration"
                     <Configuration>
                     "End"

<Configuration> ::= { <TransRule> | <TSStructureRule> |
                      <ASPTypesRule> | <PDUTypesRule> |
                       <StripSignalsRule> <Function> }*

<TransRule>    ::= "TRANSLATE"
                   [ "SIGNAL" ] <AlternativeListOfTerms>
                   <TransRuleIf>* [ <TransRuleNoIf> ]
                   "END"

<TransRuleIf> ::= "IF" <Conditions> "THEN"
                   <TransRuleNoIf> "END"

<TransRuleNoIf> ::= { "CONSTRAINT"
                      <TransRuleConstraint> |
                      "TESTSUITE"
                      <TransRuleTestSuite> }*

<TransRuleConstraint> ::= { "NAME" <Term> |
                            "PARS" <ParameterList1> |
                            "MATCH" <ParameterList1> }*

<TransRuleTestSuite>  ::= { "CONSTS" <ParameterList1> |
                            "PARS" <ParameterList2> }*

<ParameterList1> ::= <Parameter1> { "," <Parameter1> }*

<Parameter1>     ::= "$" <Number> [ "=" <Term> ]

<ParameterList2> ::= <Parameter2> { "," <Parameter2> }*

<Parameter2>     ::= "$" <Number> [ "=" <Term> ]
                     [ "/" <Term> ]

<TSStructureRule> ::= "PLACE" <AlternativeListOfTerms>
                      <TSStructureRuleIf>*
                      [ <TSStructureRuleNoIf> ]
                      "END"

<TSStructureRuleIf> ::= "IF" <Conditions> "THEN"
                        <TSStructureRuleNoIf> "END"

<TSStructureRuleNoIf> ::= "IN" <Term> { "/" <Term> }*

<ASPTypesRule>      ::= "ASP-TYPES"
```

```
                            <SequentialListOfTerms> "END"
<PDUTypesRule>          ::= "PDU-TYPES"
                            <SequentialListOfTerms> "END"
<StripSignalsRule>      ::= "STRIP-SIGNALS"
                            <SequentialListOfTerms> "END"
<Function>              ::= "FUNCTION" <Identifier>
                            <Mappings> "END"

<Mappings>              ::= <Mapping> { "|" <Mapping> }*

<Mapping>               ::= <Conditions> ":" <Term>

<Term>                  ::= <Atom> { "+" <Atom> }*

<Atom>                  ::= "$" <Number> | "@" <Number> |
                            <Text> | <Identifier> |
                            <FunctionCall>

<FunctionCall>          ::= <Identifier>
                            "(" <SequentialListOfTerms> ")"

<SequentialListOfTerms>     ::= <Term> { "," <Term> }*

<AlternativeListOfTerms>    ::= <Term> { "|" <Term> }*

<Conditions>            ::= <Condition> { "AND" <Condition> }*

<Condition>             ::= <Term> { "==" | "!=" } <Term> |
                            "TRUE"
```

For a detailed description of the semantics, see "Syntax and Semantics of the Autolink Configuration" on page 1471 in chapter 35, *TTCN Test Suite Generation*.

State Space Files

Using the command Save-State-Space, a Labelled Transition System (LTS) representing the state space generated during exhaustive exploration is saved to a file.

Syntax

The syntax of the generated LTS is (using BNF notation):

LTS	::=	‘START:’ StateId ‘LTS:’ TransList* ‘STATES:’ State*
TransList	::=	StateId ‘:’ Event ‘-’ [StateId] (‘,’ Event ‘-’ [StateId])* ‘;’
Event	::=	(Output | Input | Timeout | Internal)
Output	::=	‘o(’ Signal ‘,’ ‘"’ GraphRef ‘"’ ‘)’
Input	::=	‘i(’ Signal ‘,’ ‘"’ GraphRef ‘"’ ‘)’
Timeout	::=	‘t(’ Timer ‘)’
Internal	::=	‘x’
State	::=	‘*****’ StateId ‘*****’ Process*
Process	::=	ProcessName ‘:’ InstanceNo ‘State:’ StateName (VariableName ‘:’ Value)* ‘Input port:[’ Signal (‘,’ Signal)* ‘]’ ‘Timers:{’ Timer (‘,’ Timer)* ‘}’ Procedure*
Procedure	::=	‘Procedure’ ProcedureName ‘:’ ‘State:’ StateName (VariableName ‘:’ Value)*
Signal	::=	SignalName ‘(’ ParameterName (‘,’ ParameterName)* ‘)’
Timer	::=	TimerName ‘(’ ParameterName (‘,’ ParameterName)* ‘)’

Lexical Elements

The lexical elements used above are:

StateId	An integer denoting a system state.
GraphRef	A string denoting a graphical reference to an SDL diagram.

ProcessName	An id denoting a process type.
InstanceNo	An integer denoting a process instance.
StateName	An id denoting a state in an SDL process graph.
VariableName	An id denoting a process variable.
Value	A string denoting a value for a variable of any defined SDL type.
ProcedureName	An id denoting a procedure.
ParameterName	An id denoting a signal or timer parameter.

Example 341: A Generated LTS

A simple example of a generated LTS:

```
START:121
LTS:
2456:o(sig1(true,3),"5 P1 1 80 100")-43567;
43567:i(sig2(1),"5 P1 1 80 120")-2467,
      i(sig2(2),"5 P1 1 100 120")-98567;
121:x-2456;
2467:x-27645;
98567:x-27645;
27645:i(sig2(1),"5 P1 1 80 120")-2467,
      i(sig2(2),"5 P1 1 100 120")-98567;
STATES:
***** 121 *****
P1:1 State:idle Parent:null Offspring:null
Sender:null i:0 Input port:[ ] Timers:{ }
***** 2456 *****
P1:1 State:idle Parent:null Offspring:null
Sender:null i:5 Input port:[ ] Timers:{ }
***** 43567 *****
P1:1 State:s1 Parent:null Offspring:null Sender:null
i:5 Input port:[ ] Timers:{ }
***** 2467 *****
P1:1 State:s1 Parent:null Offspring:null Sender:null
i:1 Input port:[ ] Timers:{ }
***** 98567 *****
P1:1 State:s1 Parent:null Offspring:null Sender:null
i:2 Input port:[ ] Timers:{ }
***** 27645 *****
P1:1 State:s1 Parent:null Offspring:null Sender:ENV
i:5 Input port:[ ] Timers:{ }
```

Restrictions

Restrictions on the SDL System

The restrictions on the SDL system that can be validated with the SDL Explorer are basically the restrictions imposed by the SDL to C compilers. See "Restrictions" on page 2753 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler* for more information.

In addition to these general restrictions, the following restrictions are specific to the SDL Explorer:

- Time is not represented in the state space explored by the explorer. This makes the behavior of "Now" expressions special. In the SDL Explorer, "Now" will always return 0.
- If user-defined types with dynamic memory allocation are used, the type must be implemented as a pointer to a data area. A function for freeing the memory must also be supplied. For more information, see "More about Abstract Data Types" on page 2704 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*.
- In some cases there will be problems when the name of an entity (system, block, process, channel) is identical to another entity. A conflict will for example appear when producing names for instances in MSC diagrams.

Restrictions on Monitor Input

There following restrictions apply to monitor input:

- A parameter to a monitor command may not contain more than 1,000 characters.
- **On UNIX**, control characters of different types may terminate the validation program. `<Ctrl+C>`, `<Ctrl+D>`, and `<Ctrl+Z>` are typical characters that might terminate a validation program.

Restrictions on Dynamic Checks

There are a number of dynamic checks that are not performed at all or performed at the C level by the C runtime system. A C runtime error

will of course lead to the validation program being terminated. The following check is not made at the SDL level.

- Overflow of integer and real values are checked at the C level if the actual C system performs these checks. (These checks are likely not to be performed.)

Chapter 52 The SDL Explorer

Chapter

53 *Validating a System*

This chapters provides general information related to validation in the SDL Suite and describes the actions you perform when validating an SDL system.

For a reference to the SDL Explorer user interface, see chapter 52, *The SDL Explorer*.

How to use Autolink, a part of the SDL Explorer, is described in "Using Autolink" on page 1431 in chapter 35, *TTCN Test Suite Generation*.

Introduction

Application Areas

The SDL Explorer is a tool intended to support engineers involved in development of specifications or designs using SDL. It is designed to give the engineers a possibility to increase the quality of their work and to automate time-consuming tasks. It is focused on the following major application areas in the development process:

- It provides an automated fault detection mechanism that checks the robustness of the application and finds inconsistencies and problems in an early stage of development. This is often referred to as *validating* an SDL system. See "Validating an SDL System" on page 2417.
- When verifying the system against requirements, the Explorer provides a possibility to perform automatic verification of the requirements expressed using the MSC (Message Sequence Chart) notation. See "Verifying an MSC" on page 2430.
- When designing safety-critical or complex systems the Explorer provides a possibility to test specific properties of the design. See "Using Observer Processes" on page 2437.
- When developing TTCN test cases, the Autolink feature of the Explorer can be used to create and use MSC test purposes and to generate TTCN test cases. See "Using Autolink" on page 1431 in chapter 35, *TTCN Test Suite Generation*.

Structure of an SDL Explorer

An executable explorer is built up in the same way as a simulator. See "Structure of a Simulator" on page 2234 in chapter 50, *Simulating a System* for more information.

The same interactive monitor system as for a simulator is used, but the set of available commands differ. The graphical user interface to the explorer monitor, the *Explorer UI*, works in the same way as the Simulator UI, but the set of available command buttons differ. For a description of some other differences, see "The SDL Explorer User Interface" on page 2398.

Underlying Principles and Terms

The SDL Explorer is based on a technique called *state space exploration*, which is a well-known technique for automatic analysis of distributed systems. All state space exploration tools for SDL are based on the idea of an automatic generation of the reachable state space for the SDL systems.

For readers interested in a more detailed description of the possibilities given by state space exploration for validation of distributed systems (focused on protocols), an excellent description is given in [17], see "References" on page 2472.

Behavior Trees

The SDL Explorer operates on structures known as *behavior trees* or reachability graphs. A behavior tree is a tree structure that represents the behavior of an SDL system.

The nodes of the tree represent SDL *system states*. A system state is defined by:

- The process instances that are active
- The variable values of these processes
- The SDL control flow state of the process instances
- Any procedure calls (with local variables etc.)
- Signals (with parameters) that are present in the queues of the system
- Active timers
- Etc.

The edges between the nodes in the tree represent atomic SDL events that transfers the SDL system from one system state to another. Therefore, the edges are also called *behavior tree transitions*. They can be individual SDL statements like tasks, inputs, outputs, etc. but also complete SDL transitions, depending on how the Explorer is configured.

The size and structure of the behavior tree can thus vary and is determined by a number of Explorer options. These options affect the number of system states generated for an SDL transition, and the number of possible behavior tree transitions from a state in the tree.

State Space Explorations

The set of all system states represented by the behavior tree is called the *state space* of the system. By moving around in the behavior tree, the behavior of the SDL system can be explored and the system states reached can be examined. This is known as *state space exploration*, and it can be performed both manually and automatically.

> **Note:**
>
> The “children” of a node in the behavior tree are not generated until a state space exploration actually reaches that node, i.e., the tree is not a static structure generated when a explorer is started.

For each system state reached during state space exploration, a number of *rules* are checked to detect errors or possible problems in the SDL system. If a rule is violated, a *report* is generated to the user. By investigating the report and the system state where it was generated, the cause of the error can be determined.

States and Paths

The original start state of the system is called the *system start state*. It is the system state where the static process instances have been created but their initial start transitions have not been executed.

The *current state* is the system state that currently is under investigation. It is changed when manually navigating in the behavior tree, or when going to the system state where a report has been generated. Initially, it is set to the system start state.

The *current root* of the behavior tree can be any system state. A number of Explorer commands and features use it as a starting point of operation. Initially, it is set up to the system start state, also known as the *original root* of the behavior tree. If it is redefined, it is not possible to reach a state above the current root in the behavior tree without resetting it back to the original root.

A *path* between two states in the behavior tree can be denoted by a sequence of integers, each one indicating which transition was used to get between two states in the path. The *current path* is a path that is set up when manually navigating in the behavior tree, or when going to the system state where a report has been generated. When set up, it is the path between the current root and the current state. The current path is

changed when the Explorer moves to a state that is not part of the current path, e.g. when manually navigating to a system state outside of the current path. However, moving up and down along the current path does not change it.

Generating and Starting an SDL Explorer

There are two ways to generate and start a explorer:

- A quick way in one single step, adequate for most situations
- A more complex way in several steps, giving you complete control of the generation and start process.

In the following, the more complex way will be described first, to give a full understanding of the process. The quick way is described in "Quick Start of an SDL Explorer" on page 2395.

Generating an Explorer

The Explorer is implemented as a precompiled run-time kernel to the SDL to C Compiler. To start an Explorer for an SDL system, or a part of an SDL system, it is thus necessary to first generate an executable explorer. This is done from the Organizer.

To generate an executable explorer:

1. Select a system, block, or process diagram in the Organizer.
2. Select *Make* from the *Generate* menu. The Make dialog is opened.
3. Turn on the options *Analyze & generate code* and *Makefile*.
4. From the *Standard kernel* option menu, select *Validation*.
5. If you need to check the Analyzer options, click the *Analyze Options* button. In the dialog, set the options and click the *Set* button. For more information about these options, see "Analyzing Using Customized Options" on page 2618 in chapter 55, *Analyzing a System*.

6. Click the *Make* button.

   An explorer for the system is now generated in the current directory with the name `<system>_xxx.val` **(on UNIX)**, or `<system>_xxx.exe` **(in Windows)**, where the `_xxx` suffix is platform or kernel/compiler specific. The Status Bar of the Organizer reports the progress of the generation; the last message should be "Compiler done."

7. Open the Organizer Log window from the *Tools* menu and check that no errors occurred and that an explorer was generated.
   - If errors were found, correct them and repeat the generation process. See "Locating and Correcting Analysis Errors" on page 2624 in chapter 55, *Analyzing a System*.
   - If no explorer was generated, repeat the generation process, but click the *Full Make* button in the Make dialog instead.

Starting an SDL Explorer

An executable explorer can be run in two different modes; graphical mode and stand-alone mode (textual mode).

Graphical Mode

In graphical mode, the Explorer takes advantage of the graphical user interface and integration mechanism of the SDL Suite. A separate graphical user interface, the *Explorer UI*, is started, giving access to the monitor system through the use of menus, command buttons, etc.

To start an explorer in graphical mode:

1. Select *SDL > Explorer UI* from the Organizer's *Tools* menu. The graphical user interface of the Explorer is opened (see "The Graphical Interface" on page 2398).
2. Select *Open* from the Explorer UI's *File* menu. A File Selection Dialog dialog is opened.

   - Alternatively, click the *Open* quick button in the tool bar.
3. In the dialog, locate and select an executable explorer and click *OK*.

A welcome message is printed in the text area of the Explorer UI. The monitor system is now ready to accept commands. Please see "Supplying Values of External Synonyms" on page 2396 for some additional information that may affect the start-up.

Stand-alone Mode (Textual Mode)

In stand-alone mode, the Explorer uses the input and output devices currently defined on your computer, which provide a textual, command line based user interface. A very limited graphical support is provided when running the Explorer in this mode.

To start an explorer in stand-alone mode, the generated explorer is executed directly from the OS prompt, e.g.

```
csh% ./system_vla.val
```

A welcome message is printed on the terminal:

```
Welcome to SDL EXPLORER

Command :
```

The monitor system is now ready to accept commands. Please see "Supplying Values of External Synonyms" on page 2396 for some additional information that may affect the start-up.

> **Note:**
>
> **On UNIX**, before an explorer can be run in stand-alone mode, you must execute a command file from the operating system prompt. The file is called `telelogic.sou` or `telelogic.profile` and is located in the binary directory that is included in your `$path` variable.
> For csh-compatible shells: **`source <bin.dir>/telelogic.sou`**
> For sh-compatible shells: **`. <bin.dir>/telelogic.profile`**

Quick Start of an SDL Explorer

An explorer can also be generated and automatically started in graphical mode in one single step.

To quick start an explorer, click the *Explore* quick button in the Organizer's tool bar. The following things happen:

- An explorer is generated by using the explorer kernel that is specified in the Make dialog. (If no explorer kernel is specified, a default explorer kernel is used.)

- The graphical Explorer UI is started. If an Explorer UI with the same explorer name is already open, this UI is reused. If another Explorer UI is open, a dialog is opened where you can select to start a new UI, or to reuse one of the existing UI's.
- The generated explorer is started from the Explorer UI.

It is possible to start an explorer for a part of an SDL system (a block or a process) by selecting the block/process and then clicking on the Explore button.

Restarting an SDL Explorer

An executing explorer can be restarted from the beginning to reset its state completely:

- In graphical mode, select *Restart* from the Explorer UI's *File* menu. (This is the same as opening the same explorer again.) A confirmation dialog is opened.
- In stand-alone mode, the explorer has to be exited with the Exit command and then executed from the OS prompt again.

Supplying Values of External Synonyms

The SDL system for the explorer may contain external synonyms that do not have a corresponding macro definition (see "External Synonyms" on page 2650 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*). In that case, you will be asked to supply the values of these synonyms, either by selecting a file with synonym definitions or by entering each synonym value from the keyboard.

In stand-alone mode, the following prompt appears:

```
External synonym file :
```

Enter the name of a file containing synonym definitions, or press `<Return>` to be prompted for each synonym value.

In graphical mode, a file selection dialog is opened. Either select a file (`*.syn`) containing synonym definitions, or press *Cancel* to be prompted for each value in a separate dialog. In this dialog, the name and type of the synonym is shown together with an input text field. You can now do one of the following:

- Enter a value and click *OK*.
- Click *Default value* to get a “null” value for the synonym type entered in the input field. Accept or edit this value and click *OK*.
- Click *Cancel* to give the synonym a “null” value (without the possibility to edit the value).
- Click *Cancel all* to give the synonym and all following synonyms a “null” value.

If a synonym file is selected in the file selection dialog, this file is also used when the explorer is restarted. (If you by any chance want to use another synonym file you have to start a new Explorer UI.)

If you set the environment variable `SDTEXTSYNFILE` to a file before starting the SDL Suite, this file will automatically be used. If `SDTEXTSYNFILE` is set to “[[“ all synonyms are given “null” values.

The syntax of a synonym file is described in “Reading Values at Program Start up” on page 2651 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*.

Actions on Explorer Start-up

When an explorer is started, the static process instances in the system are created, but their initial transitions are not executed.

In some cases when the Explorer is started, a message is printed that it is not possible to generate test values for all sorts and/or signal parameters. This has to do with the automatic test value generation mechanism that is used in the Explorer. It happens if there are signals coming from the environment of the SDL system that have parameters of a sort that the test value generation cannot handle. To overcome this, define some test values for the sort that the Explorer is complaining about. See “Defining Signals from the Environment” on page 2445 for more information.

The SDL Explorer User Interface

Monitor commands in the Explorer are issued in the same way as in the Simulator, since the same monitor system is used in both tools. Also, the Explorer UI works in the same way as the Simulator UI. Please see "Issuing Monitor Commands" on page 2241 for more information.

The Explorer UI can be customized in the same way as the Simulator UI. Please see "Customizing the Simulator UI" on page 2249 for more information.

However, there are a few differences between the user interface of the Explorer and the Simulator. These differences are described below.

Activating the Monitor

The explorer's monitor system becomes active when the explorer is started, when a transition executed during navigation has completed, when an automatic state space exploration has finished, when a report with Abort action has been generated, or when an automatic state space exploration is manually stopped.

These conditions are listed in greater detail in "Activating the Monitor" on page 2299 in chapter 52, *The SDL Explorer*.

The Graphical Interface

The graphical Explorer UI is illustrated in the figure below:

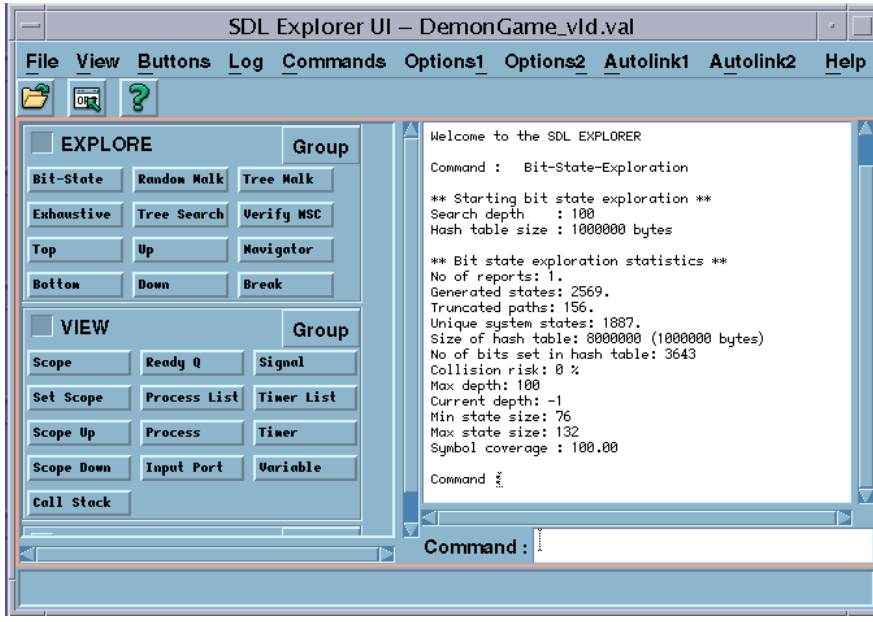


Figure 487: The Explorer UI

Since the set of available commands differ between the Simulator UI and the Explorer UI, the set of button modules and command buttons is also different. In addition, three extra menus are available in the menu bar: *Commands Menu*, *Options1 Menu* and *Options2 Menu*. The menu choices in these menus are similar to the command buttons in the sense that each of them correspond to a certain monitor command.

The Command and Watch Windows

The Command and Watch windows are also available in the Explorer UI. The difference compared to the Simulator UI is:

- In the Command window, the default commands that are executed are “List-Process -” and “Print-Trace 1”.

Navigating in the State Space

The SDL Explorer provides the possibility to interactively walk around in the behavior tree of an SDL system. This is also known as manually *navigating* in the state space. A dedicated graphical tool, the Navigator, is available in the Explorer to facilitate manual navigating. However, it is possible to use manual navigation without using the Navigator tool.

The Navigator is intended to be used in three different situations:

1. When learning how a state space exploration tool like the Explorer works, the Navigator is a convenient tool for interactively investigating the behavior tree of an SDL system.
2. When using automatic state space exploration, there is sometimes a need to start the exploration from a different starting point than the system start state of the SDL system. In this case, the Navigator can be used to walk to a suitable system state, from which the automatic exploration can be started.
3. When investigating a report generated during automatic exploration, the Navigator can be used to check the alternative behaviors that are possible on the path to the reported situation.

To open the Navigator tool, use one of the following methods:

- Select *Show Navigator* from the *Commands* menu.
- In the button module *Explore*, click the *Navigator* button.
- Enter the command Show-Navigator.

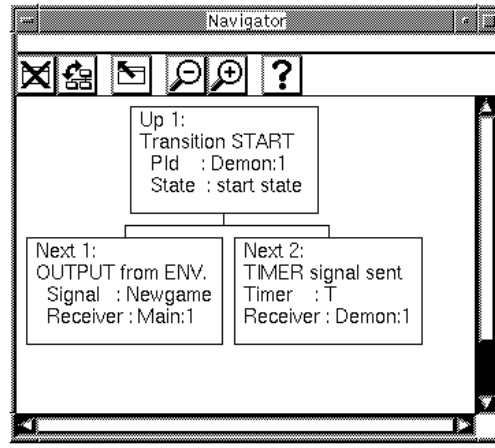


Figure 488: The Navigator

The boxes shown in the Navigator represent the behavior tree transitions leading to and from the current system state. They are labelled *Up 1* (for the transition leading to the current state) and *Next 1*, *Next 2*, etc. (for the transitions leading from the current state).

To close the Navigator, click the *Close* quick button in the tool bar.

Moving Up in the Behavior Tree

To move one level up in the behavior tree, use one of the following methods:

- In the Navigator, double-click the Up node, or select *Up 1* from the pop-up menu available on the Up node.
- In the button module *Explore*, click the *Up* button.
- Enter the command Tree-Walk 1.

To move more than one level up in the behavior tree at once:

- Enter the command Tree-Walk, followed by the number of levels to move up.

To move to the current root of the behavior tree, i.e. the top of the current path, use one of the following methods:

- In the Navigator, select *Up to top* from the pop-up menu available on the Up node.
- In the button module *Explore*, click the *Top* button.
- Enter the command Top.

Moving Down in the Behavior Tree

To see the possible Next nodes when the Navigator is not opened, enter the command List-Next. This gives a numbered list of all transitions leading from the current state.

To move one level down in the behavior tree, use one of the following methods:

- In the Navigator, double-click one of the Next nodes, or select *Goto* from the pop-up menu available on the Next nodes. This will follow the branch one step.
- Enter the command Next, followed by the number of the Next node, i.e. the number of the transition to execute.

To move more than one level down in the behavior tree at once:

- Enter the command Random-Down, followed by the number of levels to move down. For each level, a transition is chosen at random.
- Enter the command Continue-Until-Branch, or in the Navigator, select *Continue* from the popup menu available on the Next nodes. This will follow the branch several steps until there are more than one transition possible

Moving Along the Current Path

The *current path* can be seen as the path in the behavior tree that has been explored last. It is set up when going to a report (see "Going to a Report" on page 2416) and when interactively walking down the behavior tree.

The transitions making up the current path are labelled with three asterisks “***” in the nodes in the Navigator. However, no such marking is present when the transitions are listed with the List-Next command.

To move up along the current path, use the Up or Top commands as described in “Moving Up in the Behavior Tree” on page 2401 (above).

To move one level down along the current path, use one of the following methods:

- In the Navigator, double-click the Next node labelled with three asterisks “***”, or select *Goto* from the pop-up menu available on this node.
- In the button module *Explore*, click the *Down* button.
- Enter the command Down 1.

To move more than one level down along the current path at once, enter the command Down, followed by the number of levels to move down.

To move to the bottom of the current path, use one of the following methods:

- In the button module *Explore*, click the *Bottom* button.
- Enter the command Bottom.

Redefining the Current Root

The current root of the behavior tree is initially set up to the system start state. The current root is automatically redefined to the current state when using MSC verification (see “Verifying an MSC” on page 2430). It can also be redefined as an effect of changing explorer options (see “Affecting the State Space” on page 2459).

In addition, you can at any time redefine the current root to either the current state or back to the system start state. To do this, enter the command Define-Root. Select or enter *Current* to redefine the current root to the current state. Select or enter *Original* to redefine the current root to the system start state.

Going to a System State

When using the Explorer it is quite common that there is a need to go to a specific system state, for instance to be able to start an automatic state space exploration from this point. This section describe some possibilities to in an efficient way get to the wanted system state, called the *target state* below.

Using Manual Navigation

In many cases the simplest way is to use the Navigator tool or the navigation commands to interactively traverse the path to the target state. Manual navigation is described in "Navigating in the State Space" on page 2400.

Returning to an Already Reached State

It is possible to return to a state that has been reached in an earlier stage when using the Explorer. Three methods will be discussed:

- Using Path Commands
- Using the Command Log
- Using MSC Trace.

The benefit of the first two techniques is that the exact same target state will be reached. The drawback is that these techniques will not work as soon as either the SDL system or the state space options have been changed (see "State Space Options" on page 2466).

The benefit of the MSC technique is that it is less vulnerable to changes in the state space options or in the SDL system. The drawback is that the exact same target state may not be reached. We only know that the path to the reached system state will satisfy the generated MSC trace.

Using Path Commands

To go to the target state using path commands:

1. When in the target state, enter the command Print-Path. The path from the root state to the current state is printed. The path is a se-

quence of integers indicating which transitions must be chosen to get to the current state.

2. At a later stage, enter the command Goto-Path, followed by the path printed above.

Using the Command Log

To go to the target state using the command log:

1. Before navigating to the target state, select *Start Command Log* from the *Log* menu, or enter the command Command-Log-On. Specify a file name on which all subsequent monitor commands will be stored.
2. Navigate to the target state.
3. Select *Stop Command Log* from the *Log* menu, or enter the command Command-Log-Off. The command logging is stopped and the file is closed.
4. Return to the same state in which the command log was started.
5. Execute the commands stored in the file by selecting *Include Command Script* from the *Commands* menu, or enter the command Include-File. Select or specify the earlier file name.

Using MSC Trace

To go to the target state using MSC trace:

1. When in the target state, generate an MSC trace from the root state to the current state. Enter the command MSC-Log-File, followed by a file name.
2. Return to the root state by using the Top command.
3. Go to the end of the MSC trace by verifying the MSC. See "Verifying an MSC" on page 2430.

Using an MSC

If an MSC is created that describes the events leading to the target state, verifying this MSC gives a possibility to go to a system state that satisfies the MSC in an efficient way. It does not matter if the MSC is manually created, generated from the Simulator or from the Explorer itself (as discussed in "Using MSC Trace" on page 2405, above). However, the exact same target state may not be reached by this method. We only know that the path to the reached system state will satisfy the generated MSC trace.

Using a User-Defined Rule

If the target state can be described in terms of process states, variable values, etc., a convenient way to get to a state that satisfies the description is to use a user-defined rule (see "Using User-Defined Rules" on page 2455).

To go to a target state using a user-defined rule:

1. Define the rule describing the target state (see "Managing User-Defined Rules" on page 2456).
2. Define the report action for user-defined rules reports to be Abort (see "Report Action" on page 2464). This will cause an automatic exploration to stop as soon as a state is reached that satisfies the rule.
3. Start an automatic state space exploration (see "Using a Default Exploration" on page 2417).
4. Go to the state where the rule was satisfied and a report was generated (see "Going to a Report" on page 2416).

The benefit with this method is that it is fast and efficient, especially if the target state is on a considerable depth in the state space. The drawback is that sometimes there are shorter paths to the target state than the one that was automatically generated.

Tracing, Logging and Viewing Facilities

In the Explorer, the same kind of commands as in the Simulator are available for tracing the execution, logging the user interaction and examining the system. There are a few differences, described below.

Tracing the Execution

Textual Trace

In the Explorer, the same type of printed trace information is available for executed transitions as in the Simulator; see "Textual Trace" on page 2255. Unlike the Simulator, however, there is no command to start continuously printing the textual trace; instead, a command must be explicitly used whenever a trace is wanted.

- To print a textual trace for the transitions leading to the current state, enter the command Print-Trace, followed by the number of transitions to trace. That is, Print-Trace 1 prints the trace for the latest transition. The same information as for a full trace during simulation is printed.
- By default, the command Print-Trace 1 is executed in the Command window of the Explorer UI, i.e., a continuous trace is in practice available in graphical mode.

Graphical SDL Trace

Graphical trace of SDL symbols in the source GR diagrams is available. The graphical trace in the Explorer selects all symbols that were executed in the transition leading to the current state. This is different from the Simulator, where GR trace selects the **next** symbol to be executed.

- To enable or disable continuous graphical trace, enter the command SDL-Trace. This command toggles the graphical trace; the current state of the trace is printed after the command is executed. When the trace first is enabled, an SDL Editor is opened as soon as the next transition is executed.
- In the Explorer UI, the graphical trace can be controlled from the *Commands* menu. The command *Toggle SDL Trace* toggles the trace between enabled and disabled.

MSC Trace

MSC trace enables tracing of executed events in an MSC Editor. When the trace first is enabled, an MSC Editor is opened, showing the events executed up until the current state, and the current path is set up. After that, the trace is continuously updated in the MSC Editor as transitions are executed. This means that events are **added** when you navigate down the behavior tree, the selected event is **changed** when you navigate up, and the MSC is **redrawn** when you move outside the current path.

- To enable or disable continuous MSC trace, enter the command MSC-Trace. This command toggles the trace; the current state of the trace is printed after the command is executed.
- In the Explorer UI, the MSC trace can be controlled from the *Commands* menu. The command *Toggle MSC Trace* toggles the trace between enabled and disabled.

The MSC trace from the current root to the current state can also be saved on a log file, which later may be opened from an MSC Editor. To save such an MSC log, enter the command MSC-Log-File, followed by the file name. The MSC log file should be given the file extension `.mpr`.

Before an MSC trace is started, you may define what types of events that will be traced. See "MSC Trace Options" on page 2465 for more information.

Logging the User Interaction

The interaction between the user and the Explorer can be logged on file in exactly the same way as in the Simulator. See "Logging the User Interaction" on page 2278 in chapter 50, *Simulating a System* for more information.

Examining the System

The current state of the system can be examined in the same way as in the Simulator. The View commands available in the *View* module of the Explorer UI are generally the same ones as in the *View* module of the Simulator UI.

Current Process and Scope

Some of the commands used for examining the system operate on a specific process instance, the *current process*, identified by the current *scope*. A scope is a reference to a process instance, a reference to a service instance if the process contains services, and possibly a reference to a procedure instance called from this process/service (the *current procedure*).

The scope is automatically set by the explorer to the process instance that executed in the transition leading to the current system state. You may change the scope if you would like to examine another process, service or procedure instance.

- To print the current process/service scope, click the *Scope* button in the *View* module, or enter the command Scope.
- To set the current process/service scope:
  - Click the *Set Scope* button in the *View* module, or enter the command Set-Scope. This command takes one parameter, a process instance, and optionally if the process contains services, a second parameter which specifies a service name.
  - Select or enter the name of a process instance.
  - If the process instance contains services, select or enter the name of a service instance.
  - The scope is set to the specified process/service, at the bottom procedure call.
- To print the procedure call stack for the process/service instance defined by the current scope, click the *Call Stack* button in the *View* module, or enter the command Stack.
- To change the procedure scope within the current process/service scope, you can move the scope one step up or down in the procedure call stack. Click the *Up* or *Down* button in the *View* module, or enter the command Scope-Up or Scope-Down. Going up from a service leads to the process containing the service. To go down in a service within a process, select or enter the name of the service instance.

Commands to Examine the System

The available commands are shortly described below. See "Examining the System" on page 2264 in chapter 50, *Simulating a System* for more information.

- To list the process instances in the ready queue, enter the command List-Ready-Queue, or click the *Ready Q* button.
- To print overview information about process instances, enter the command List-Process, or click the *Process List* button.
- To examine a process instance, enter the command Examine-PId, or click the *Process* button. The process instance must be specified as the first parameter.
- To list all signal instances in the input port of a process instance, enter the command List-Input-Port, or click the *Input Port* button. The process instance must be specified as the first parameter.
- To examine a signal in the input port of a process instance, enter the command Examine-Signal-Instance, or click the *Signal* button. The process instance must be specified as the first parameter.
- To list all currently active timers, enter the command List-Timer, or click the *Timer List* button.
- To examine a timer instance, enter the command Examine-Timer-Instance, or click the *Timer* button.
- To examine a variable of a process instance, enter the command Examine-Variable, or click the *Variable* button. The process instance must be specified as the first parameter.

Performing Automatic State Space Explorations

This section describes how to perform an automatic state space exploration and how to examine the results. The application areas in which automatic state space exploration are used are further described in "Validating an SDL System" on page 2417, "Verifying an MSC" on page 2430, "Using Observer Processes" on page 2437 and "Using Autolink" on page 1431 in chapter 35, *TTCN Test Suite Generation*.

In the Explorer, three types of automatic state space explorations can be used, implemented as different algorithms:

- Bit state exploration, an efficient algorithm for reasonably large SDL systems.
- Random walk, a simple algorithm that can be used for very large SDL systems.
- Exhaustive exploration, an algorithm suited only for small SDL systems.

The characteristics of these algorithms are further described in "Configuring the SDL Explorer" on page 2458. They have the following in common:

- They start from the current system state, which means that you may have to navigate to a suitable start state before the exploration is started.
- They explore the state space down to a certain depth from the start state, to avoid exploring an infinite state space forever.

The performance and results of a state space exploration are also highly dependent on how the state space is configured. This is discussed in "State Space Options" on page 2466.

The most important monitor commands concerning state space explorations are available in the *Explore* module in the Explorer UI.

Executing an Exploration

The different types of explorations are started in the following way:

- To start a bit state exploration, enter the command Bit-State-Exploration, or click the *Bit-State* button.
- To start a random walk, enter the command Random-Walk, or click the *Random Walk* button.
- To start an exhaustive exploration, enter the command Exhaustive-Exploration (there is no button for this command).

> **Note:**
>
> The button *Verify MSC* starts a bit state exploration, configured to suit MSC verification. This is further described in "Verifying an MSC" on page 2430.

When the exploration is started, a message is printed stating the options used for this exploration type (see "Configuring the SDL Explorer" on page 2458):

```
** Starting bit state exploration **
Search depth    : 100
Hash table size : 1000000 bytes


** Starting exhaustive exploration **
Search depth : 100


** Starting random walk **
Depth       : 100
Repetitions : 100
```

By default, the exploration continues until it is finished, i.e., until the state space has been fully explored according to the exploration options. During the exploration, a status message is repeatedly printed after a certain number of transitions or states have been generated.

> **Note:**
>
> Depending on how an exploration is configured, it may take a considerable amount of time to finish!

To stop an exploration manually, click the *Break* button in the Explorer UI, or hit `<Return>` in stand-alone mode. A stopped exploration may

be continued by issuing the same exploration command again. You are then asked whether to continue the exploration from the state where it was stopped, or restart the exploration from the same start state as before.

When the exploration is finished or stopped, some exploration statistics are printed (see "Interpreting Exploration Statistics" on page 2413). By default, a tool called the *Report Viewer* is also opened (see "Examining Reports" on page 2414).

The Explorer always returns to the start state of the exploration when it is finished or stopped.

Rules Checked During Exploration

During state space exploration, a number of rules are checked to detect errors or possible problems in the SDL system. If a rule is satisfied, a report is generated to the user.

The rules are used to find design errors, typically caused by unexpected behaviors of the SDL system. Often the errors are caused by events happening at the same time in different parts of the system, for example when a signal is received from the environment of the system at the same time as a timer expires. So-called signal races are often part of the error situations that are found.

Apart from the predefined rules, an additional rule can be defined by the user to check for other properties of the system. See "Using User-Defined Rules" on page 2455 for more information.

Interpreting Exploration Statistics

The different exploration algorithms print somewhat different statistics. The important statistics to note are the following:

- `No of reports: x`
  The number of error situations found. How to investigate the reports are described in "Examining Reports" on page 2414.

- `Truncated paths: x`
  The number of times the exploration reached the maximum search depth. The execution path is at that stage truncated and the exploration continues in another state. If this value is greater than 0, parts

of the state space have not been explored. However, this is a normal situation for SDL systems with infinite state spaces.

- `Collision risk: x`
  For bit state explorations, the risk (in percent) for collisions in a hash table used to represent the generated system states (see "Bit State Exploration Options" on page 2460). This value should be **very** small, 0-1%; otherwise, the size of the hash table may have to be increased. If collisions occur, some execution paths may be truncated by mistake.

- `Current depth: x`
  The search depth reached at the moment the exploration was finished or stopped. If this value is -1, the exploration finished by itself. If the depth is greater than 0, the exploration was stopped. In this case, it may be continued from this depth, as described in "Executing an Exploration" on page 2412.

- `Symbol coverage: x`
  The percentage of the SDL symbols in the system that have been reached during the exploration. If this value is less than 100, parts of the system have not been explored.

What actions to take depending on the printed statistics is discussed in "Validating an SDL System" on page 2417.

Examining Reports

When an exploration has been performed, the reported error situations should be examined. A dedicated graphical tool, the Report Viewer, is available in the Explorer to facilitate the report examination. However, it is possible to examine the reports without using the Report Viewer.

The Report Viewer is by default automatically opened when an exploration has been performed. To open the Report Viewer manually, either select *Show Report Viewer* from the *Commands* menu, or enter the command Show-Report-Viewer.

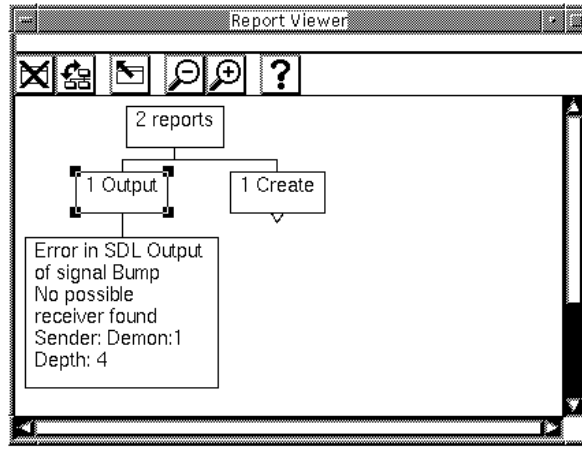


Figure 489: The Report Viewer

The nodes in the Report Viewer are structured in three levels and show, from top to bottom:

- The total number of generated reports
- The different types of reports (errors) and the number of reports of that type
- The actual reports with error message and trace information, and the exploration depth where the error was generated (this level of the tree is by default collapsed).

To close the Report Viewer, click the *Close* quick button in the tool bar.

To list the reports when the Report Viewer is not opened, enter the command List-Reports, which prints a numbered list of all reports.

Changing the Displayed Structure

Generally, to expand or collapse a node in the Report Viewer, double-click the node or select *Expand* or *Collapse* from the popup menu available on the nodes. This works for the top node and the report type nodes; for report nodes, see "Going to a Report" on page 2416 (below).

To show the whole report structure, select *Expand Substructure* from the popup menu available on the top node. To collapse the whole struc-

ture, select *Collapse* from the same pop-up menu, or double-click the expanded top node.

To switch between a tree structure and a list structure, click on the *Structure* quick button. The list structure makes it possible to easier see the different reports and report types when a large number of reports have been found.

Going to a Report

When "going to a report," the Explorer goes to the system state where the report was generated. You can then examine the reported situation further.

To go to a report using the Report Viewer:

1. Expand the report structure to show the desired report node.
2. Double-click the report node, or select *Goto* from the pop-up menu available on the report node.

To go to a report using monitor commands:

1. List the reports by entering the command List-Reports, and note the number of the desired report.
2. Enter the command Goto-Report, followed by the report number.

After going to a report, the Navigator tool is updated and the current path is set up. You can walk along the path to the error by using the Navigator; see "Moving Along the Current Path" on page 2402.

By default, an MSC Editor is also opened, showing the MSC trace from the current root to the state where the report was generated.

Validating an SDL System

This section describes how to use the automatic state space exploration facilities in the Explorer to look for inconsistencies and design errors in an SDL system. The idea is essentially to test the robustness of the application, the responses to unexpected situations. Essentially the validation is an attempt to answer questions like:

- What happens if a user does not press the buttons in the order assumed by the designer?
- What happens if the scheduling algorithm of the operating system that supports the application is changed?
- What happens if the environment happens to send an input to the system at the same time as a timer expires?

...and all other questions the designer never ever would imagine.

It is assumed that the SDL system is of moderate size and complexity; techniques for validating large SDL systems are described in "Validating Large Systems" on page 2422.

Using a Default Exploration

When you are to use the Explorer to try finding errors in a new SDL system for the first time, you are advised to start a bit state exploration using the default options.

To validate a system opened in the Explorer:

1. If you already have executed commands for the opened explorer, you should reset the explorer. Enter the command Reset, or click the *Reset* button in the *Explore* module. This is especially important if you earlier have loaded an MSC into the Explorer.
2. You should also make sure you use the default state space and exploration options. Enter the command Default-Options, or click the *Default* button in the *Explore* module.
3. Start a bit state exploration (see "Executing an Exploration" on page 2412). Let the exploration run for at least 10-20 minutes.
4. If the exploration has not finished by itself, stop it manually (see "Executing an Exploration" on page 2412). The Report Viewer is

opened and the exploration statistics is printed. Note especially what the symbol coverage is.

5. Use the Report Viewer to go to each of the reported situations (see "Examining Reports" on page 2414). Navigate along the current path to the report and use the tracing and viewing facilities to examine the report.

6. If you find errors in the system, you may decide to correct them immediately. In that case, generate a new explorer for the corrected system and rerun the validation, as described above. Otherwise, you should check if the validation is to be considered finished (see below).

Determining if the Validation is Finished

When all reports have been checked and the found errors possibly have been corrected, the next question arises: When are we finished validating the system? To answer this question, look at these aspects:

- What was the symbol coverage reported in the statistics after the automatic exploration?
- Did the exploration finish by itself or was it stopped by the user?

The following possibilities now exist:

1. The symbol coverage is 100% and the exploration finished by itself.

   All symbols have been executed and furthermore most orderings of the possible actions have been tested. In this case it is probably not worthwhile continuing the validation; you may consider it finished.

   However, not all orderings of possible actions have been tested, since the search may have been truncated, collisions may have occurred in the hash table, and more orderings are possible by configuring the state space exploration differently. If you want, you can change the explorer options and start a new exploration (see "Using Advanced Validation" on page 2421 and "Configuring the SDL Explorer" on page 2458).

2. The symbol coverage is 100% but the exploration was manually stopped.

In this case, it may still be worthwhile to continue the exploration until it finishes by itself. More reports may be generated, as there are still orderings of the possible actions that have not been executed.

3. The symbol coverage was less than 100%.

   Parts of the system have never been reached during the exploration. In this case, the validation cannot be considered finished, even if the exploration finished by itself. The reasons and possible solutions for low symbol coverage are discussed next.

Handling Low Symbol Coverage

If the symbol coverage after an exploration is 100%, all parts of the system have been executed at least once. If the symbol coverage is less than 100%, the possible reasons why parts of the state space have not been reached are listed below.

- The exploration was manually stopped before all symbols were reached.

  In this simple case, you should continue the exploration until it finishes by itself.

- The test values were inappropriate.

  Test values are used to define the set of possible signals from the environment. The automatically generated test values may not suit all SDL systems. This may for example cause the execution to never execute one branch of a decision statement. To overcome this problem, redefine the test values for the appropriate signal parameter. For more information on test values, see "Defining Signals from the Environment" on page 2445.

- The exploration was pruned after a report.

  In most cases the Explorer will *prune* the exploration of a particular path as soon as a report has been found, i.e., the exploration will not continue beneath the state in question. If you have examined such a report and has decided not to do anything about it, the Explorer will still prune the search when it finds the report the next time. To overcome this problem, change the report action for this particular report type from *prune* to *continue*. See "Configuring the SDL Explorer" on page 2458 for more information.

- Some parts of the system are, in fact, unreachable.

  If some parts of the SDL system are not reachable at all, it may be an indication that there is a design error in the system.

- There are problems with timer expirations.

  The explorer is by default configured in a way that tries to reduce the size of the state space. It will always try to execute internal actions (e.g. tasks, decisions, internal input and outputs) before any timers are allowed to expire. The assumption is that the system will always execute fast enough to ensure that no timers will expire (the timers may of course expire when waiting for input from the environment). To make a more complete test of this type of situation, see "Using Advanced Validation" on page 2421.

- The search depth was too small.

  The default search depth is 100. This may not be enough for some systems, e.g. a system with a very long initialization phase. In some cases, it is possible to overcome this problem simply by increasing the search depth (see "Configuring the SDL Explorer" on page 2458). However, the techniques discussed in "Validating Large Systems" on page 2422 are often more suitable.

- The state space is too big.

  Many SDL systems of reasonable complexity quite simply have state spaces that are too big; it is not possible to explore the entire state space in one exploration. Characteristic for this situation is a low symbol coverage, truncated paths, and either manually stopped exploration or a high (>10%) collision risk. This situation is discussed in "Validating Large Systems" on page 2422.

To find out which parts of the system that have not been reached, a tool called the Coverage Viewer is used. To start the Coverage Viewer, select *Show Coverage Viewer* from the *Commands* menu, or enter the command Show-Coverage-Viewer. If the symbol coverage was less than 100%, the Coverage Viewer will display a tree structure representing the parts of the system that have not been executed.

Using Advanced Validation

The default options for the state space exploration, in particular the options that define the structure of the state space, are optimized to give good results for a first validation of a system. They are intended first of all to test for internal inconsistencies in the SDL system and to get a good process graph coverage. This assumes a reasonably “nice” environment, i.e., the environment only sends signals when nothing can happen internally in the system.

This has the benefit of reducing the size of the state space while still preserving a good process graph coverage. The drawback is that some error situations are never detected. One particular class of errors that never will be detected using the default options can be characterized as signal races caused by signals sent from the environment, or timer expirations that happen at the same time. An example is a situation where a communication protocol ends up in an inconsistent system state when two connect requests are sent to the different access points at the same time.

To detect these types of errors it is necessary to change the options and perform a second set of explorations for the SDL system. The suitable settings of the options are called *advanced options.* When using these values for the options, the state space will get very large for most SDL systems. It is thus usually not possible to get a complete coverage of the state space, even if some of the techniques described in “Validating Large Systems” on page 2422 have been used. To anyway be able to get good results, the best strategy is to use the random walk algorithm when exploring the state space. See “Using Random Walk Exploration” on page 2428 for more information.

To set advanced options, click the *Advanced* button in the *Explore* module. In stand-alone mode, you have to enter a number of commands to achieve the same result; see “Setting Advanced Options” on page 2471 for information on which commands to use.

For a more in-depth explanation of the state space options, see “State Space Options” on page 2466.

Validating Large Systems

This section discusses various techniques that are useful when designing and validating large SDL systems. A large system is, in this context, a system that has a state space that is too large to be completely explored using one automatic state space exploration. The techniques are pragmatic and intended to give a reasonable chance of finding any errors even though the complete state space is not searched.

The following techniques are discussed:

- *Decomposed Exploration*
- *Using MSCs to Limit the Search*
- *More Efficient Bit-State Exploration*
- *Reducing the State Space Size*
- *Using Random Walk Exploration*
- *Incremental Validation*.

Decomposed Exploration

The idea when using decomposed explorations is to use a number of reasonably small explorations instead of one big exploration. Quite often the behavior of an SDL system can be divided into a number of “phases” or “features.” The idea is to explore each of these phases or features separately. The benefit with this approach is that it is a lot easier to explore the different phases separately than trying to explore the combination of all phases. The drawback is that errors that are caused by an interaction between different phases or features are not found. However, for large SDL systems it is sometimes the only possible method that at least can give a complete symbol coverage.

The process of finding which and how many partial explorations that are necessary is a combination of an iterative process and a planning issue where the possible features and phases that can be subject to a partial exploration are identified. If an incremental design process is used it is often possible to use the different iterations to guide the choice of partial explorations; compare with “Incremental Validation” on page 2429.

A common strategy used to find the needed partial explorations is essentially the following:

1. Start an exploration from the system start state.
2. Check all reports and correct the errors in the system. Generate a new explorer and make another exploration.
3. When all found reports have been fixed, check the symbol coverage. If the coverage is 100%, the validation is finished; otherwise, continue with the next step.
4. Use the Coverage Viewer to check which parts of the SDL system that need more testing.
5. Go to a suitable system state and start a new exploration from there.
6. Repeat the process until the symbol coverage is 100%.

There are two issues of this strategy:

- Where to start each partial exploration.
- How to limit each partial exploration.

Where to Start a Partial Exploration

The problem of identifying where to start a new exploration is of course system dependent and requires knowledge of the SDL system. The tool to use in this case is the Coverage Viewer, which shows exactly what parts of the SDL system that have been executed during the exploration and what parts have not been executed. Once a system state has been chosen the next issue is how to get there in the Explorer. There are number of possible ways to do this; see "Going to a System State" on page 2404.

How to Limit a Partial Exploration

The next problem is to limit each partial exploration to the intended part of the state space. There exists a number of factors which can be used to influence the extent of an exploration:

- The search depth
- The signals from the environment
- User-defined rules

The search depth is the simplest limiting factor to use. By reducing the search depth, e.g. to 10 or 20, the size of the exploration will of course be considerably reduced compared to the default depth of 100.

By changing the list of signals that can be sent from the environment it is possible to control which parts of the system that will be exercised by an exploration. For example, if we are interested in testing the data transfer phase of a connection-oriented protocol specification, a good strategy would be the following:

- Go to a system state where the connection is established.
- Define the signals from environment to be only the signals relevant for the data transfer, and start the exploration. For a description of how to define and remove signals from the list of signals that can be sent from the environment, see "Defining Signals from the Environment" on page 2445.

User-defined rules also give a possibility to limit the extent of an exploration. Define a rule that matches the system states where the exploration should be pruned and check that the report action for user-defined rules is to prune the search. For example, the rule `state(initiator:1)=idle` would prune the exploration whenever the initiator process entered the state *Idle*. User-defined rules are described in "Using User-Defined Rules" on page 2455.

Using MSCs to Limit the Search

Another possibility that sometimes is useful to control the exploration of the state space is to use MSCs to guide the exploration. This is particularly useful for SDL systems with a design that uses restrictions on the possible behavior of the environment of the system. It might, for example, be known that the signals A, B and C always will come in this order from the environment of the system. In this case it is not interesting to analyze what will happen if the signals will come in a different order.

An MSC can be loaded to guide the search by using the command Load-MSC. Once an MSC is loaded, both interactive navigation in the state space, e.g. by using the Navigator, and automatic exploration will only search the parts of the state space that correspond to the loaded MSC.

This means that if you want to go back to normal exploration, you have to clear the loaded MSC by using the commands Clear-MSC or Reset.

Note how the test values are used when an MSC is loaded. It is allowed to leave out parameters to messages in the MSC. If a parameter is left out on a signal from the environment, the test values are used to determine the parameter values that are actually sent to the system. This is a useful feature when using MSCs to limit the search. See section "Verifying an MSC" on page 2430 for more details.

Another useful hint when using MSCs is to always use system level MSCs to guide the state space exploration. A system level MSC will allow a larger part of the state space to be explored than a block or process level MSC.

An MSC loaded into the Explorer must comply with some requirements; see "Requirements for MSC Verification" on page 2436.

More Efficient Bit-State Exploration

The bit-state search uses a hash value based algorithm to store the state space that is traversed. Unfortunately the computation of hash values from a system state is an expensive operation and most of the execution time in a bit-state search is spent calculating hash values. The execution time for the hash algorithm is in most situations proportional to the size of each system state. The max and min system state size used by the hash algorithm is included in the statistics printed after each bit state search and should be checked if the search is slow. (See "Bit-State-Exploration" on page 2301).

If the size of a system state is big (> 10,000 bytes) the bit state execution of the explorer will be fairly slow. In these cases it might be worthwhile to try to optimize the performance by reducing the state size that the explorer uses when computing hash values. This can be done by informing the explorer to skip a number of variables when computing hash values. The explorer includes a command Define-Variable-Mode that is intended for this purpose. (See "Define-Variable-Mode" on page 2324.) For example the command:

```
define-variable-mode monitor subscrTab skip
```

will make the explorer skip all `subscrTab` variables in `monitor` processes.

A typical example of where this feature is useful is if the system includes a big array (or other big data structure) that is initialized at the start up of the system and that after the initialization is known to be con-

stant in the part of the state space that is explored. The correct way to take advantage of this in the explorer is to:

1. Go to a system state where the array is initialized. (See "Going to a System State" on page 2404 for more info about how to navigate in the state space.)
2. Redefine the root to the current state. (See "Define-Root" on page 2319.)
3. Change the mode of the table variable to "Skip".
4. Start the bit-state exploration.

Using this strategy it is possible to considerably increase the performance of the explorer.

Another situation where the variable mode can be changed to "Skip" is when there are variables in the system that is known not to have any influence on the dynamic behavior of the system. See "Variables Not Influencing the Dynamic Behavior" on page 2428.

Reducing the State Space Size

There is a number of ways to reduce the state space that is necessary to explore by using knowledge and assumptions about the SDL system. Usually this is based on the fact that the state space of an SDL system contains various "sub state spaces" that are equivalent except for some detail, which is not very interesting for the purpose of the validation. Some examples of such details are:

- The value of local variables
- The number of instances of process types
- The size of large data structures.
- Variables that do not influence the dynamic behavior.

Local Variable Values

An example of the way local variable values influence the size of the state space is the following: Consider a situation where a process contains an integer variable that counts the number of times a particular signal comes from the environment, and then replies with this number when requested to do so from the environment. It is obviously not espe-

cially interesting to try to investigate the behavior of the SDL system for all possible values of this local variable. Instead a reasonable set of values should be selected and the state space exploration guided by this selection.

A user-defined rule (see "Using User-Defined Rules" on page 2455) provides an efficient means to reduce the size of the state space by putting restrictions on variable values. In the example above a reasonable restriction might be that we only would like check what happens the first three times the variable is increased. A rule that expresses this is:

```
proc:1->var < 4;
```

Once this rule is defined and the report action for user-defined rule violation is set to Prune (which is the default), only the interesting parts of the state space are explored.

Number of Process Instances

Another issue is the number of process instances that are used for each process type. If the number is large and all of them do the same thing, for example by modeling different connections in a connection oriented protocol, it is probably not very useful to try to explore the combination of all instances. Instead, it is better to restrict the number of instances allowed in the exploration. This can be achieved with the command Define-Max-Instance (see "Maximum Number of Instances" on page 2470). If preferred, it is also possible to use a user-defined rule or an observer process to achieve the same result.

Size of Large Data Structures

A third area where the explorer performance is reduced is when large data structures, e.g. arrays, are used in the SDL system. A large data structure has two unfavorable effects on a state space exploration:

- The size of the reachable state space increases extremely rapidly as the size of the data structure increases.
- The efficiency of the bit state algorithm is decreased as the size of system states increase. Essentially the time to compute a new system state is linear to the size of the system states.

A good idea in this context is to, whenever possible, try to reduce the size of any large data structures in the SDL system before performing validation. Another possibility is to skip the variable when computing

hash values as described in "More Efficient Bit-State Exploration" on page 2425.

Variables Not Influencing the Dynamic Behavior

In many situations an SDL system contains a number of variables that does not have any impact on the dynamic behavior of the system. Essentially all variables that does not (directly or indirectly) have any influence on the path taken through a decision or the expression used when computing the receiver of a signal in output/RPC call will not influence the dynamic behavior of the system.

These variables can safely be ignored when performing a state space search. This can be accomplished by instructing the explorer to skip these variables using the Define-Variable-Mode command. (See "Define-Variable-Mode" on page 2324.) This will in many cases drastically reduce the size of the state space that the explorer needs to search and is an efficient way to improve the performance of the explorer.

Note that implicit variables like Sender/Parent/Offspring are also considered as variables in this respect. In particular Sender can be of interest to skip if it is not used, since it may change value every time a signal is received.

As an example, if Sender is not used in a process 'p' the following command will make the explorer ignore the Sender implicit variable when comparing two system states:

```
define-variable-mode p Sender skip
```

Using Random Walk Exploration

In some situations it is not possible to use the more elaborated techniques described in this section to cope with the problem of validating large SDL systems. The time and resources available for the validation may simply be too limited. A possible strategy to use when validating very large SDL systems is to use the random walk exploration strategy instead of the bit state algorithm.

The reason is that the random walk algorithm gives a possibility to get a partial exploration of the state space that is randomly chosen. Furthermore, the symbol coverage of the exploration is determined only by how long the exploration is allowed to run. The drawback with the algorithm is that if it is allowed to run for a long time, so that significant

parts of the state space already have been covered, there is no mechanism to avoid that already explored paths are explored once more.

How to start a random walk exploration is described in "Executing an Exploration" on page 2412. The random walk exploration algorithm is further described in "Random Walk Options" on page 2461.

The best way to get an idea of what has been tested when using random walk is to use the Coverage Viewer to check the symbol coverage. Even if this is not the same as the coverage of the system state space, it will show if there are large portions of the system that have not been reached by the exploration.

Incremental Validation

A common way to develop large SDL specifications and designs in practice is to use an incremental development strategy. First a base functionality is implemented and then various features are added in an incremental fashion. When this type of development process is used, a good way to plan the validation of the system is to let the different increments define the state space explorations that should be performed.

First a number of state space explorations are executed with different start states, and perhaps different test values. Together these explorations should give a good process graph coverage of the SDL system representing the base functionality.

For each increment that is added, a number of additional explorations is performed that will cover the new features in the SDL system.

It is also probably worthwhile to define command scripts that automatically can execute the various explorations that should be run to achieve a good process graph coverage. This makes it possible to run all of the various explorations in an straight-forward way for each new increment that is added to the system.

Verifying an MSC

MSC verification is one the major application areas for the SDL Explorer. This section describes how to use the Explorer to get started with MSC verification. It also gives some ideas of how to organize MSCs to be able to use common initialization MSCs and shows how to use batch files to achieve an efficient regression testing of an SDL system using MSC verification.

The first prerequisite for MSC verification is of course that we have an MSC that describes some desirable behavior that can be used to check the SDL system against. This MSC can be interactively created using the MSC Editor as part of a requirement analysis, but it is also possible to use MSCs created as execution traces in the SDL Simulator or the Explorer itself as input to the MSC verification.

It is worth noticing that the MSC does not have to be a process level MSC. It is possible to use MSCs where the MSC instances correspond to SDL blocks and systems, and even mixed MSCs where some instances correspond to processes and other to blocks.

There are some requirements on MSCs to be used for MSC verification; see "Requirements for MSC Verification" on page 2436.

The characteristics of the MSC verification algorithm is further described in "MSC Verification Options" on page 2462.

Basic MSC Verification

To verify an MSC for a system opened in the Explorer:

1. If necessary, go to a system state that corresponds to the start of the MSC to verify. If the MSC describes events from the start of the system, go to the system start state. You may have to reset the explorer first, especially if you already have an MSC loaded.
2. To start the MSC Verification, click the *Verify MSC* button in the *Explore* module, or enter the command Verify-MSC.
3. The MSC that you want to verify has to be specified. Either select it in the File Selection Dialog that appears (in graphical mode), or enter the name of the file on the command line (in stand-alone mode).

Note: Illegal characters in path name

Please note that verifying an MSC fails if the path name of the MSC contains "(" or ")".

4. A bit state exploration adapted to suit MSC verification is performed. After the exploration statistics, the result of the MSC verification is presented. If it was possible to find an execution trace that was consistent with the MSC, the text

   ```
   ** MSC <MSC name> verified **
   ```

   is printed, where <MSC name> is the name of the MSC that was checked. If it was not possible to find a consistent execution trace, the following text is printed:

   ```
   ** MSC <MSC name> NOT VERIFIED **
   ```

5. If the MSC was not verified, check the generated reports using the Report Viewer. There will be a number of "MSCViolation" reports. These reports identify the execution paths which violate the MSC, i.e., paths that contain events that are not part of the MSC. You may investigate these reports by using the method described in "Examining Reports" on page 2414.

6. If the MSC was successfully verified, there will be a "MSCVerification" report (there may also be a number of "MSCViolation" reports, but they can be discarded). You do not have to go to this report; the Explorer automatically goes to the state where the MSC was verified. This means that the current path is set up automatically.

7. To verify another MSC from the same start state, go to the top of the current path. It is now possible to directly start a new MSC verification, as described above (you do not have to reset the explorer).

Note:

When MSC verification is started, the current root of the behavior tree is redefined to the current state. This feature is used in the next section, "Verifying a Combination of MSCs Using High-Level MSCs" on page 2433. (It also means that you may have to reset the explorer to be able to reach the system start state again.)

Converting Instances Before Verification

Before an MSC is loaded into the Explorer for verification, it is possible to perform *instance conversion* of the MSC. Instance conversion will convert the name of an instance to another name.

This is useful if you want to verify some, but not all, instances in an MSC with an SDL system. For instance, you may have an MSC describing a complete system but an SDL system for only a part of the system. In this case, you can convert the not wanted instances to be considered as environment in the Explorer, without changing the MSC.

Note that if you have more than one instance representing the environment, the environment instances must be separated using channel names.

Instance conversion is performed before an MSC is loaded into the Explorer by entering the command "Define-Instance-Conversion FromString ToString" for each instance name to be converted. All instance conversions can be listed by entering the command List-Instance-Conversion, and all instance conversion can be cleared by entering the command Clear-Instance-Conversion.

Example 342 ----------

Consider an MSC with three process instances A, B and C. The SDL system specifies the behavior for instance A, but not for B or C. Before verifying the MSC, B can be converted to "channelB" and C to "channelC", where channelB is the existing SDL channel that will be used for communication between the existing A process and the non-existing process B, and channelC is the existing SDL channel that will be used for communication between the existing A process and the non-existing process C.

This is accomplished by entering the Explorer commands:

```
Define-Instance-Conversion B "channelB"
Define-Instance-Conversion C "channelC"
```

The MSC can now be verified.

Verifying a Combination of MSCs Using High-Level MSCs

The high-level MSC that are available in MSC'96 provide a very convenient possibility to describe how many small MSCs are combined to form a larger use case or scenario. The Explorer support verification of high-level MSCs.

As an example, consider a situation where we have an MSC "init" that will describe some initialization phase that is needed to set up the SDL system to some "connected" state from where two features are accessible. These features are described by the MSCs "datatrans" and "finish". If this would be a communication protocol, the "init" might be a connection establishment, and "datatrans" and "finish" successful data transfer and connection release. This situation could be described using the high-level MSC in Figure 490.

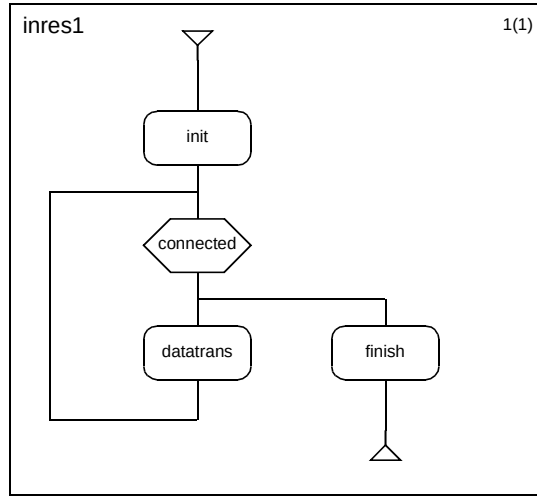


Figure 490 A high-level MSC

To check this combination of MSCs simply verify the "inres1" high-level MSC and the explorer will generate one MSC verification report for each sequence of "leaf MSCs" that can be verified. In this case there will be reports for "init, finish", "init, datatrans, finish", "init, datatrans, datatrans, finish", etc. until the max depth for MSC verification has been reached.

State of the SDL Explorer after MSC Verification

When an MSC verification has been done, the current state of the explorer is different depending on how many MSC verification reports has been found:

- If 0 or more than 1 MSC verification report has been generated the current state of the explorer is the system state where the MSC verification was started from.
- If exactly 1 MSC verification report was generated the current state is the state where this report was found.

This is in many cases useful when using MSCs to navigate to a specific system state, especially in combination with command scripts as described in the next section. Note that if the MSC that is verified is an "old style" MSC without high-level MSCs or MSC reference expressions then there will always be at most one MSC verification report and this type of MSCs is thus best when using MSCs for navigation.

Using Batch MSC Verification

An efficient test strategy when incrementally developing SDL systems is to use regression testing. A set of MSCs describe the requirements on the SDL system and new MSCs are incrementally added to the set when new features are implemented in the SDL system. Each time a new feature is implemented the resulting system should be tested against all the old MSCs as well as the new ones, to make sure that no new errors have been introduced.

To accomplish this using the Explorer the most efficient way is to use the command script facility in the Explorer. A command script is simply a text file containing a number of Explorer commands, usually one command per line. A command file can be loaded into the Explorer by selecting *Include Command Script* from the *Tools* menu, or by entering the command Include-File.

Example 343: Batch MSC Verification ––––––––––––––––––––––––––

A command script that when loaded will perform MSC verification for some requirements described as MSCs may look like:

```
log-on msc.log
verify-msc inres1.mrm
reset
verify-msc init2.msc
verify-msc inres2.mrm
quit
```

The command Log-On is used to store the output from the verification on the file `msc.log`.

––

Note in Example 343 how the MSC init2 was used to set up the start state for the verification of the high-level MSC inres2.

Verifying Message Parameters

When verifying an MSC, the parameters of the messages in the MSC can sometimes be crucial and need to be verified, and sometimes be unimportant for the behavior in question. To support this, the verification of MSCs in the Explorer allows three different levels of matching of message parameters:

- No parameters are given for the message.
- Only some of the parameters are given.
- All of the parameters are given.

If no parameters are given, all possible actual parameters are accepted. If the signal is sent from the environment to the system, the parameter values that are defined using the test value facility are used by the Explorer when exploring the state space of the system.

When only some of the parameters are to be given, only the given parameters are checked during the exploration. The notation used to show that a specific parameter should be ignored during the verification is to simply leave out this parameter in the parameter list. For example, if only the second and fifth parameter should be used during the verification the parameter list would be “,2,,,true” in the MSC. Trailing commas can be left out, so if a signal has five parameters and only the first

two are to be verified, the parameter list might be “1,2” which would ignore the last three parameters.

When only some of the parameters are given for a signal from the environment, the rest of the parameters are taken from the test value definitions when executing the signal output during state space exploration.

If all parameters are given, they are of course all checked.

Requirements for MSC Verification

The MSCs that can be loaded into the Explorer must comply with the following rules:

- It must be possible to map each MSC instance to either the environment, a channel to/from the environment, the entire SDL system, a block, or a process. This mapping is done by matching the name of the MSC instance with the names of the corresponding SDL entities. However, the name of an MSC instance can be changed before verification, see “Converting Instances Before Verification” on page 2432.
- PId values are not allowed as parameters to MSC messages from the environment of the SDL system. PId values are allowed on internal messages and messages to the environment, but the values are not checked during the exploration.
- If the MSC is on process level, only one static instance of each process type is allowed in the MSC. There is no limit to the number of dynamically created MSC instances.
- Only the following events/symbols are interpreted in an old-style MSC. All other events are ignored or will not be accepted by the Explorer.
  - input
  - output
  - set
  - reset
  - timeout
  - create
  - stop
  - global MSC reference symbol without substitution and gates
  - condition

Note:

Conditions are interpreted as a synchronization point if ***Define-Condition-Check*** is set to “on”, otherwise they are ignored. For more information, see ***“Synchronizing Test Events with Conditions” on page 1444 in chapter 35, TTCN Test Suite Generation***. Set, reset and timeout events on MSC instances representing SDL channels to the environment are only accepted by the Autolink test generation commands Generate-Test-Case and Translate-MSC-Into-Test-Case. For all other Explorer commands which load an MSC, timer events on environment instances are ignored and the Explorer generates a warning.

- Only the following symbols are allowed in a high level MSC:
  - start symbol
  - end symbol
  - MSC reference symbol without substitution and gates.
  - condition symbol (ignored during verification)
  - connection point.
- In MSC reference symbols it is allowed to use MSC reference expressions with the operators:
  - alt
  - par
  - seq
  - exc
  - opt
  - loop

Using Observer Processes

The purpose of an observer process is to make it possible to check more complex requirements on the SDL system than can be expressed using MSCs. The basic idea is to use SDL processes (called *observer processes*) to describe the requirements that is to be tested and then include these processes in the SDL system. Typical application areas include feature interaction analysis and safety-critical systems.

To be useful, the observer processes must be able to inspect the SDL system without interfering with it and also generate reports that convey the success or failure of whatever they are checking.

To accomplish this, three features are included in the Explorer:

- The observer process mechanism.

  By defining processes to be observer processes, the Explorer will start to execute in a two-step fashion. First, the rest of the SDL system will execute one transition, and then all observer processes will execute one transition and check the new system state.

- The assert mechanism.

  The assert mechanism enables the observer processes to generate reports during state space exploration. These reports will show up in the list of generated reports in the Report Viewer. The details of the assertion mechanism is discussed in "Using Assertions" on page 2457.

- The *Access* abstract data type.

  The purpose of the Access abstract data type is to give the observer processes a possibility to examine the internal states of the other processes in the system. Using the Access ADT it is possible to check variable values, contents of queues, etc., without any need to modify the observed processes. See "The Access Abstract Data Type" on page 2440 for more details.

A simple observer process is illustrated in Figure 491.

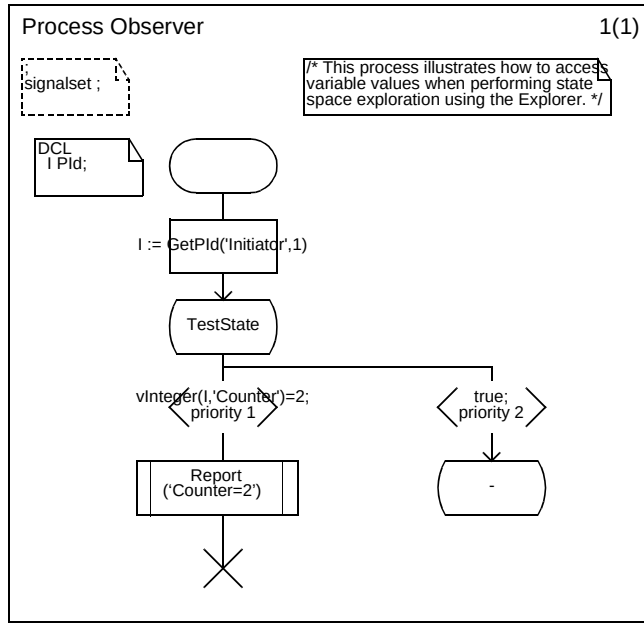


Figure 491: A simple observer process

This process will check if the variable Counter in the Initiator process ever becomes equal to 2.

Two characteristics for the observer processes are:

- the use of continuous signals with tests that use Access operator to check the internal state of other processes, and
- the use of assertions to report the result.

To use observer processes:

1. Create the observer processes in the SDL Editor. You should place the observer processes in a special block that you include in the diagram structure of the SDL system. In this block, you also need to specify an INCLUDE directive for the Access ADT:

   ```
   /*#INCLUDE 'access.pr' */
   ```

2. In the generated explorer, define each observer process by using the command Define-Observer, followed by the name of the process type. All instances of the process type will now become observer processes.

3. Perform a state space exploration. If an assertion defined in an observer process is satisfied, an "Assertion" report is generated. To simplify the observer processes, an "Observer" report will also be generated whenever there is an observer process that cannot execute.

In some cases the Observer reports are not convenient and they can then be turned off with the Define-Report-Continue (that will cause the exploration to continue past a reported situation) and the Define-Report-Log command (that can be used to turn off the logging a s specific report type, e.g. Observer reports)

The Access Abstract Data Type

The Access abstract data type is an ADT intended to be used together with observer processes to make it possible to access the internal state of other processes from an observer process. The ADT is defined in the file `access.pr` that resides in the `ADT` directory together with the rest of the ADTs supplied together with the SDL Suite. Unlike the rest of the ADTs the Access ADT is a special purpose ADT that only works with the Explorer kernel.

The Access ADT defines a number of SDL operators. The signatures of these operators are defined as follows:

```
GetPID : CharString, integer -> PId;
  /* Returns the PId value of a process instance

     par 1: the name of the process type
     par 2: instance number of the process instance
  */

ActivePID : integer -> PId;
  /* Return the PId of the process that executes.
     Returns NULL if no process executes.
     Observer processes are not taken into account.

     The integer parameter is a dummy parameter
     needed since operators must have parameters. */

GetState  : PId -> CharString;
  /* Returns the name of the current state of a
```

```
          process instance (or previous if the process
          is not in a state)

        par 1: the pid of the process instance */

GetNoOfInst: charstring -> integer;
   /* Returns the number of instances for a
      particular process type

      par 1: the name of the process type */

 terminated: PId -> boolean;
   /* Returns true if the process instance is
      terminated otherwise false

      par 1: The PId value of the process instance */

 GetProcedure: PId -> charstring;
   /* Returns the name of the procecure a process
      instance currently has called. If no procedure
      is called 'none' is returned

      par 1: The PId value of the process instance */

 InProcedure: PId, CharString -> boolean;
   /* Returns true if a process instance currently
      executes in a specific procedure, otherwise
      false

      par 1: The PId value of the process instance
      par 2: The name of the procedure */

 GetNoOfSignals: PID -> integer;
   /* Returns the number of signals currently in the
      input port of a process.

      par 1: The PId value of the process instance */

 GetSignalType: PId, integer -> charstring;
   /* Returns the name of the signal type for a
      signal in the input port of a process instance

      par 1: The PId value of the process instance.
      par 2: A number (>=1) identifying the signal */

 InQueue: PId, charstring -> boolean;
   /* Returns true if a process instance currently
      has a signal of a specific type in its input
      port queue

      par 1: The PId value of the process instance.
      par 2: The name of the signal type. */

 /* Variable access functions. These functions return
    a variable value that corresponds to one of the
```

```
    variables of a process instance as specified by
    the parameters.

      par 1: The PId value of the process instance.
      par 2: Variable name  */

v : PId, charstring -> integer;
vInteger : PId, charstring -> integer;
v : PId, charstring -> real;
vReal : PId, charstring -> real;
v : PId, charstring -> boolean;
vBoolean  : PId, charstring -> boolean;
v : PId, charstring -> Character;
vCharacter : PId, charstring -> Character;
v : PId, charstring -> Time;
vTime : PId, charstring -> Time;
v : PId, charstring -> Duration;
vDuration : PId, charstring -> Duration;
v  : PId, charstring -> Charstring;
vCharstring : PId, charstring -> Charstring;
v : PId, charstring -> PId;
vPId : PId, charstring -> PId;

EnvOutput : CharString -> Boolean;
  /* Returns true if the transition currently
     executing is caused by a signal from the
     environment with a specified name, otherwise
     false

     par 1: The name of the signal type */
```

The Access ADT also includes a utility procedure called *Report* that if called from an observer process will generate an Assertion report in the explorer. The procedure takes a charstring as parameter and this is the string that will be presented in the Assertion report.

An example of the usage of some of the operators is:

```
P := GetPId( 'Initiator', 1 ),
CS := GetState( P )
```

This example assigns the PId value of the first Initiator process to the PId variable P, and then assigns the name of the state of this process to the charstring variable CS.

An example of a statement that will access the variable value for one of the variables of another process instance is:

```
LocalVar := v(P,'Var')
```

In this example we assume that we have a PId variable `P` that identifies a process that has a variable `Var` of type for which a `v` operator is de-

fined. The statement will access the value of `Var` and assign it to the local variable `LocalVar`.

The operators for accessing the value of a variable are given in two versions for each predefined simple type: the `v` operator and the `vXXX` operator, where XXX is the name of the type. They are equivalent and the only time there is a need to use the `vXXX` operator is when it is not possible to resolve by context which of the `v` operators that is intended.

To access variables of sorts that are syntypes to the predefined simple types, the `v` operator for the corresponding predefined simple type can be used.

Accessing variable of structured types, enumeration types and user-defined types is a bit more complex. There are two possible ways to do it. Either define the `v` operator for the type in question, or use the **#CODE** operator and access the variable value using a C macro `XVV`.

Example 344: Structured Types in Observer Processes

Consider the following structure definition:

```
newtype MSDUType
  struct
    id IPDUType;
    num Sequencenumber;
    data ISDUType;
endnewtype MSDUType;
```

A `v` operator for this type can be defined as:

```
newtype MSDUTypeAccess
literals NotUsedMSDUTypeAccess;
operators
  v /*#NAME 'XVNAME(MSDUType)'*/ :
    PId, charstring -> MSDUType;
/*#ADT (H)
#TYPE
#define MSDUType #SDL(MSDUType)
*/
endnewtype MSDUTypeAccess;
```

Once this definition is in place, variable values for the complex data type can conveniently be accessed using the new `v` operator. Note also that it is possible to access the values of the fields of the structure in a simple way:

```
LocalVar := v(P,'MSDUVar')!id
```

If the type is one of the types passed as values, according to the table in "Parameter Passing to Operators" on page 2676 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*, XVNAME should be substituted to XVNAME2.

However, if the values of the complex type only is accessed in a few places, it is possible to access them directly using the #CODE operator as illustrated in the following example:

```
LocalVar := #CODE('XVV(#SDL(P),"Var",#SDL(MyType))')
```

In this example we assume that we have a PId variable `P` that identifies a process that has a variable `Var` of type `MyType`. The statement will access the value of `Var` and assign it to the local variable `LocalVar`.

Defining Signals from the Environment

A problem common to all state space exploration techniques is related to the treatment of the environment of the SDL system under analysis. As an example, consider the situation during state space exploration where a signal with an integer parameter can be received from the environment. Since there is an infinite number of integer values, there will be an infinite number of successors of the current system state: one where the parameter value is 0, one where the parameter value is 1, etc.

This is obviously a situation that is not acceptable when performing state space exploration. The SDL Explorer allows three different strategies to avoid situations like this:

1. Create a closed system by specifying the environment of the system using SDL. This will solve the problem but introduces a new one; it is necessary to create an SDL model of the environment.

2. Specify the signals that can be sent from the environment to the system. This is a simple way to avoid the problem. By enumerating the signals with their parameters that the environment can send, a finite branching is guaranteed at each system state in the state space.

3. Use an MSC to guide the state space exploration. Since the MSC defines what signals the environment can send and their ordering, a limited part of the state space can be explored.

The second strategy is the most common and the test value feature of the Explorer is designed to make it easy to define the signals from the environment.

Test Values

When the Explorer is started a list of signals is automatically computed that will be used as the possible signals from the environment during state space exploration. The signal list is generated based on the concept of test values. Test values can be defined for data types and for signal parameters. When generating the signal list the Explorer checks for each signal that can come from the environment which test values are defined for its parameters (or for the parameter data types). It then generates one signal instance for each combination of test values for the parameters.

Each time the Explorer is in a state where input from the environment is possible during state space exploration, the list of signals defined by the test values is consulted.

The default test values for the simple data types are:

Data Type	Default Test Values
`Integer`	-55, 0, 55
`Boolean`	true, false
`Real`	-55, 0, 55
`Natural`	0, 55
`Character`	‘a’
`Charstring`	“test”
`Duration`	0
`Time`	0
`PId`	Environment PId
`Bit`	0, 1
`Octet`	00, FF
`Bit_string`	’01’B
`Octet_string`	’00FF’H

For other data types, test values are determined according to the following:

- Enumerated types: All values in the type
- Subranges of the predefined data types: All values in the range
- Structures: All combinations of the test values of the individual fields
- Arrays: All combinations of the test values of the component type.
- Ref types: NULL + pointers to the test values for whatever the Ref points to.
- Own types: NULL

- ORef types: NULL

Test Values Restrictions and Options

Two restrictions are posed on the computed test values:

- If the number of test values for a data type or signal parameter exceeds a maximum number, randomly chosen test values will be generated.
- If the number of signal instances for a particular signal type exceeds a maximum number, randomly chosen signal instances will be generated for this signal type.

Two commands exist for setting options related to the above restrictions:

- To define the maximum number of test values for any data type or signal parameter, enter the command Define-Max-Test-Values, followed by the number of test values. The default is 10.
- To define the maximum number of signal instances for any signal type, enter the command Define-Max-Signal-Definitions, followed by the number of signal instances. The default is 10.

Note:

These options affect the state space; see "Affecting the State Space" on page 2459.

Defining and Listing Test Values

The default test values are defined to be useful for a large number of applications, but they sometimes need to be modified. In some cases there are unnecessarily many test values and to enhance the performance of the state space exploration some test values can be cleared. In other cases the automatic test value generation cannot handle some of the data types used, so the test values must be manually defined.

Changing the test values are therefore only needed if you would like to fine-tune the behavior of the Explorer, or if the signals from the environment have parameters that are of a user-defined or unusual data type.

Note:

Changing test values affects the state space; see "Affecting the State Space" on page 2459.

Test values can be defined and cleared on three "levels": on data types, on individual signal parameters, and on signal instances. When test values are defined or cleared, the list of signals from the environment is regenerated. You are recommended to define test values either on data types and individual signal parameters, or on signal instances; do not combine both these methods.

The monitor commands concerning test values are available in the *Test Values* module in the Explorer UI.

Test Values for Data Types

The following commands operate on the test values for a data type (sort).

- To define a new test value for a sort, enter the command , or click the *Def Value* button. The parameters are the sort and the value. Example:

```
integer 20
```

- To list the new test values defined for all sorts, enter the command List-Test-Values, or click the *List Value* button.
- To clear all test values for a sort, enter the command Clear-Test-Values, or click the *Clear Value* button. As parameter, you either specify the sort, or '-' which means **all** sorts.

Test Values for Signal Parameters

The following commands operate on the test values for individual parameters to a signal.

- To define a new test value for a signal parameter, enter the command Define-Parameter-Test-Value, or click the *Def Par* button. The parameters are the signal, the ordinal number of the signal parameter, and the value. Example:

```
Define-Parameter-Test-Value Score 1 -5
```

- To list the new test values defined for all signal parameters, enter the command List-Parameter-Test-Values, or click the *List Par* button.
- To clear all test values for a signal parameter, enter the command Clear-Parameter-Test-Values, or click the *Clear Par* button. As parameter, you specify the signal and the ordinal number of the signal parameter. You may use '-' for the parameter number, which means **all** signal parameters, or just '-' for the signal, which means **all** signal parameters for **all** signals.

Test Values for Signal Instances

The following commands operate on the test values for a specific signal instance.

- To define a new set of test values for a signal instance, enter the command Define-Signal, or click the *Def Signal* button. The parameters are the signal and an optional set of values for the parameters. Multiple Define-Signal commands may be used to define several signal instances of the same signal type, but with different values. Example:

```
Define-Signal Test 10 'hello' true
Define-Signal Test -5 'bye'
```

Note:

The signals defined using this command are cleared when the signal list is regenerated, e.g. if a test value is defined for a sort or a signal parameter.

- The command Extract-Signal-Definitions-From-MSC analyzes basic MSCs in textual form (with suffix `.mpr`) and extracts all signals sent from the environment axes to the system axis. If a signal definition is found which does not already exist, it is added automatically by calling Define-Signal.
- To list all currently defined signal instances, enter the command List-Signal-Definitions, or click the *List Signal* button.
- To clear all test values for a signal type, enter the command Clear-Signal-Definitions, or click the *Clear Signal* button. As parameter, you specify the signal, or '-' which means **all** signals.

Saving Test Values

The current set of test values can be saved on file and later be recreated by reading in the file again. The file will contain monitor commands that recreates the saved set of test values and discards any other test values.

To save the test values, enter the command Save-Test-Values, followed by a file name. To read in the saved test values again, enter the command Include-File, followed by the file name.

Validating Systems That Use the Ref Generator

The Ref generator (see "The Ref Generator" on page 111 in chapter 2, *Data Types, in the SDL Suite 6.2 Methodology Guidelines*) is used to create pointer structures to be used in SDL systems. The Explorer supports the Ref generator, but imposes some restrictions on the usage of it due to the special requirements caused by state space exploration.

Variables that are defined to be Ref's to something can be used in two ways, either as a pointer to some other variable or as a pointer to a dynamically allocated memory area. Both ways of using Ref types are supported by the Explorer.

To handle dynamically allocated data areas the Explorer creates a special data structure as part of each system state. This data structure is a list of all data areas allocated by `Alloc` (the Ref operator that allocates a new data area) and data areas allocated in external C code (see "Validating Systems with External C Code" on page 2452). The list contains for each data area in addition to the area itself information about e.g. the sort of the data area and the size of the data area. Whenever the Explorer copies a system state, the list of dynamically allocated data areas is also copied and all Ref variables are set up corresponding to the new copy of the list.

Some restrictions/simplifications are needed when using Ref sorts in the Explorer:

- Variables may not be defined to be of the VoidStar or VoidStarStar sorts, since the Explorer needs to know the sort and size of what is pointed to. This is not known for VoidStar and VoidStarStar sorts.
- A simplification is made when comparing two system states for equivalence (both in exhaustive exploration and in the hash function used by bit state exploration): Two Ref variables are considered equivalent if the data they are pointing to are equivalent. This may in some cases prune the search in situations where it should not have been pruned. Note that equivalence tests in the SDL system works correctly, if two Ref variables are compared using '=' they are considered the same only if they contain the same pointer value.

Note that the handling of pointers in the Explorer introduces a significant overhead that unfortunately reduces the number of transitions per second that is executed by the Explorer.

When performing state space exploration, the Explorer checks the usage of Ref variables when copying system states and reports several different types of problems including:

- memory leaks, and
- pointers to released or never allocated memory.

For more information about the reports see "REF Errors" on page 2374 in chapter 52, *The SDL Explorer*.

Validating Systems with External C Code

the SDL Suite allows the usage of external C code together with an SDL system and this is also true for the Explorer. In many cases it is possible to directly use the Explorer on a system that uses external C code. However, due to the special requirements of state space exploration, some restrictions must hold for the external C code, and some modifications may have to be done to the external code to make it functions properly with the Explorer.

To be able to perform a state space exploration it must be possible for the Explorer to make a complete copy of a system state, including all data structures that are implemented directly in C code. The Explorer must also be able to modify each copy of a system state separately. This has some implications:

- variables defined in C code cannot be handled by the Explorer,
- C unions may not contain pointers, data types implemented by pointers (like the SDL types string and bag) or SDL PIds, and
- some restrictions on the usage of pointers are needed since the C pointers in SDL are treated like Ref types (see "Validating Systems That Use the Ref Generator" on page 2451).

If there are variables in C code, this will not be detected by the explorer. It may appear as if the Explorer works, but the variables defined in C code will not be copied when the Explorer copies a system state. When the value of a variable is changed by an action performed in one system

state, this value will change the value for all system states that the Explorer currently handles. This implies e.g. that when the Explorer backtracks during an automatic exploration to test more possible successors of a particular system state, the values of variables defined in C may be different from the values they had the previous time the system state was visited and the state space exploration will not be correct.

In order to be able to copy a system state, the Explorer must have exact information about the sort of all data areas in the system to be able to copy e.g. pointer-based data structures correctly. One consequence of this is that the Explorer cannot support the C union sort if the union may contain pointer-based sorts, since the Explorer cannot know the current sort of the union and thus cannot deduce whether to treat the union as a pointer or not. SDL PIds are also treated specially in the Explorer and can also not be part of a C union.

Pointers are frequently used in C code and when used together with the SDL Suite they are treated as the (nonstandard) SDL type Ref. The Explorer handles the Ref types in a particular way (see "Validating Systems That Use the Ref Generator" on page 2451) and the restrictions on variables of this sort also applies to the usage of C pointers in data type in external C code.

When using dynamic memory allocation in extern C code some special additions are needed for the Explorer to work properly. This is needed since the Explorer keeps a list of all dynamically allocated data areas as part of each system state. If an external C function allocates memory, the Explorer must be informed about the data area that was allocated, and the same holds when a C function releases memory. This is accomplished by calling two functions from the C code:

```
extern void UserMalloc (void *data);
extern void UserFree (void *data);
```

UserMalloc should be called when a data area has been allocated, and UserFree should be called immediately before the data area is released. Both functions should have a pointer to the data area as parameter.

The purpose of UserMalloc is to insert a new element into the list of dynamically allocated data areas that is maintained by the Explorer. Note that there is no need to tell the Explorer what sort of data was allocated or its size. This is handled automatically by the Explorer simply by finding the SDL entity (e.g. a variable) that points at the data area and assuming that the sort and size given by this entity is correct. If no SDL

entity can be found that points to the data area, this is considered to be an error and a Explorer report is generated.

The purpose of the UserFree function is to inform the Explorer that a data area has been released, and thus should be removed from the list of dynamically allocated data areas.

There exists a special C macro XVALIDATOR_LIB that can be used to check in external C files if the code is compiled together with the Explorer kernel. It is thus possible to only include the calls to UserMalloc/UserFree when the C code is compiled together with the Explorer using this macro, as in the following example:

```
...
v = malloc( 10 );
#ifdef XVALIDATOR_LIB
UserMalloc( (void *)v );
#endif
```

Using User-Defined Rules

In the Explorer, you may define a user-defined rule to be used during state space exploration to check for properties of the encountered system states. If a system state is found for which the user-defined rule is true, a report will be generated. Note that only one user-defined rule may be defined at a time.

Different Usages

There are three different situations in which a user-defined rule is useful:

- To verify properties of the SDL system.

  A user-defined rule describes properties of system states. By using an automatic state space exploration, it is thus possible to verify the existence of system states that satisfy the specified properties. If the state space is small enough to allow a complete exploration it is also possible to verify that the state space does not contain any system state with the specified property.

- To search for specific system states.

  A user-defined rule makes it possible to go to a specific system state in the state space without the need to use the navigating commands of the explorer monitor. By describing the desired state with a rule and using an automatic state space exploration, you can go directly to the report that satisfied the rule. In this case, the report action for the user-defined rule report should be set to Abort.

- To reduce the state space to be explored.

  For many SDL systems, the state space can be very large or even infinite, which makes it difficult to perform a state space exploration effectively. However, in many cases the state space contains large subspaces that for some reason are not interesting to explore. For instance, they may be equivalent to other parts of the state space except for the value of one particular variable. In such cases, a user-defined rule can be used to restrict the exploration by defining system states that are considered to be uninteresting. When such a state is encountered, the exploration is truncated and continued in another node.

Examples of Rules

An example of a rule that checks a system property is:

```
exists P:Proc | P->var=12;
```

which is true for all system states where there exists a process of type "Proc" with a variable "var" that is equal to 12.

A simple example of a rule that searches for a system state is:

```
state(initiator:1)=disconnected;
```

which is true for all system states where the process instance "initiator:1" is in the state "disconnected".

A more complex example of such a rule is:

```
state(Game:1)=Winning and
sitype(signal(Game:1))=Probe
```

which is true for all system states where the state of the process instance "Game:1" is equal to "Winning" and the type of signal to be consumed by the same process instance is "Probe".

An example of a rule that reduces the state space is:

```
(Game:1->Count > 2) or (Game:1->Count < -2)
```

which is true for all system states where the absolute value of the variable "Count" in the process instance "Game:1" is greater than 2.

For a full description of the features and syntax of user-defined rules, see "User-Defined Rules" on page 2376 in chapter 52, *The SDL Explorer*.

Managing User-Defined Rules

To define the user-defined rule, select *Define Rule* from the *Commands* menu, or enter the command Define-Rule, followed by the definition of the rule.

To clear the user-defined rule, enter the command Clear-Rule.

To print the definition of the current user-defined rule, enter the command Print-Rule.

To evaluate the user-defined rule in the current system state, i.e. to check whether the rule is satisfied, enter the command Evaluate-Rule.

Using Assertions

Like most other run-time libraries to the SDL to C Compiler, the Explorer library gives the user a possibility to define his own run-time errors or assertions. An assertion is a test that is performed at run-time, for example to check that the value of a specific variable is within the expected range. Assertions are described by introducing `#CODE` directives with calls to the C function `xAssertError` in a TASK. See the following example.

Example 345: Assertion in C Code ––––––––––––––––––––––––––

```
TASK '' /*#CODE
#ifdef XASSERT
  if (#(I) < #(K))
    xAssertError("I is less than K");
#endif
*/ ;
```

––

In the SDL Explorer, the assertions are checked during state space exploration. Whenever `xAssertError` is called during the execution of a transition, a report is generated. The advantage of using this way to define assertions, as opposed to using user-defined rules, is that in-line assertions are computed much more efficiently by the explorer than the user-defined rules.

The `xAssertError` function, which has the following prototype:

```
extern void xAssertError ( char *Descr )
```

takes a string describing the assertion as parameter and will produce an SDL run-time error similar to the normal run-time errors. The function is only available if the compilation switch XASSERT is defined. For the standard libraries this is true for all libraries except the Application Library.

Configuring the SDL Explorer

This section describe the various possibilities available to control the behavior of the Explorer using the options that can be defined for different features. The available options are grouped into a number of categories; each category and option will be described later in this section.

Managing Options

Each option can be set using a monitor command, usually named something similar to “Define-<option>”. Most options can also be set from the menus *Options1* and *Options2* in the Explorer UI. The monitor command and menu choice associated with an option is listed together with the description of the option.

If the options are changed during a session with the Explorer, you will be asked whether to save the options when you exit or restart the currently executing explorer. If you save the options, the new values will be stored in a file named `.valinit` **(on UNIX)**, or `valinit.com` **(in Windows)**, in the directory from where the SDL Suite was started. This file will automatically be loaded the next time the Explorer is started from the same directory, thus restoring the previous options.

Some monitor commands operate on all the options:

- To print a list of all options and their current values, select *Show Options* from either the *Options1* or *Options2* menu, or enter the command Show-Options. (A few of the options described here are not listed.)
- To set all options to their predefined default values, click the *Default* button in the *Explore* module, or enter the command Default-Options. Note that this also clears all reports.
- To set all options to their initial values, i.e. the values set when the explorer was started, click the *Reset* button in the *Explore* module, or enter the command Reset.

Note:

This command also resets the explorer completely and is equivalent to restarting the explorer from scratch. To just set the options to their initial values without resetting the explorer:

1. Set the options to their default values. See above.
2. Read in the file `.valinit` **(on UNIX)**, or `valinit.com` **(in Windows)**; see above. Select *Include Command Script* from the *Commands* menu, or enter the command Include-File.

Affecting the State Space

Some of the options affect, directly or indirectly, the size of the state space and the structure of the behavior tree. This can only be done while being in the current root of the behavior tree, since the whole structure of the tree may be affected. If such an option is changed when the explorer is **not** in the current root of the behavior tree, you have two choices: either to change the current system state back to the current root, or to redefine the current root to the current system state.

In this case, the following dialog is opened:

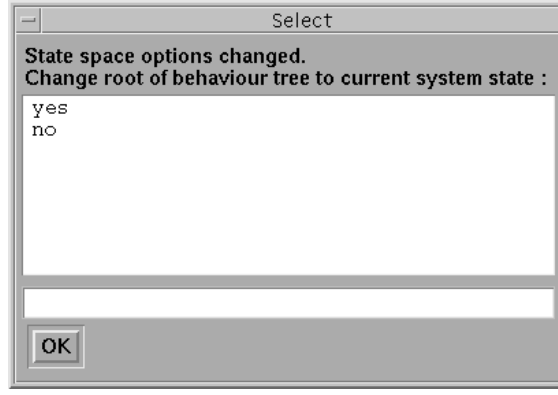


Figure 492: Changing the current root

To change the root to the current system state, select *yes* and click *OK*. (In stand-alone mode, enter **`yes`**)

To keep the current root and move back to it, select *no* and click *OK*. (In stand-alone mode, enter `no`)

> **Note:**
>
> It is **not** possible to cancel this operation, i.e. you **have to** either change the current root or the current system state.

Bit State Exploration Options

Bit state exploration is an efficient automatic state space exploration algorithm for reasonably large SDL systems (for a reference, see [17]). It performs a depth-first search through the state space and uses a bit array to store the states that has been traversed during the search.

Every time a new system state is generated during the search, two hash values are computed from the system state. The bit array is checked:

- If both of the positions indicated by the hash values are already set, the state is considered to have been previously visited. The search of this particular path in the state space is pruned, and the search backs up to a previous system state and continues elsewhere.
- If both of the positions are not set, the state is a new state that has not been previously visited. Both position in the bit array are then set and the search continues with the successor states.

Search Depth

The search depth is the maximum depth the Explorer will explore a particular execution path in the state space. When this depth is reached, the search is truncated and the search backs up to a previous system state.

- Default value: 100
- Command: Define-Bit-State-Depth
- Menu choice: *Options2: Bit-State: Depth*

Hash Table Size

The size of the bit array used as hash table is an important factor defining the behavior of the bit state exploration. The reason is that each time a new state is checked by comparing its hash values with previous hash values there is a risk for collision. The bigger the hash table is, the smaller the collision risk is.

- Default value: 1,000,000 (bytes)
- Command: Define-Bit-State-Hash-Table-Size
- Menu choice: *Options2: Bit-State: Hash Size*

Random Walk Options

Random walk is an automatic state space exploration algorithm that can be useful for very large SDL systems. It performs a depth-first search through the state space by selecting transitions to execute at random.

When the maximum search depth is reached during such a “random walk,” the search is restarted from the original state again and a new random walk is performed. However, there is no mechanism to avoid that already explored paths are explored once more, i.e. a system state may be visited a large number of times.

Search Depth

The search depth determines how many transitions will be executed before the search is pruned and restarted from the beginning again.

- Default value: 100
- Command: Define-Random-Walk-Depth
- Menu choice: *Options2: Random: Depth*

Repetitions

The number of times the random walk search will be repeated from the start state before the exploration is finished.

- Default value: 100
- Command: Define-Random-Walk-Repetitions
- Menu choice: *Options2: Random: Repetitions*

Exhaustive Exploration Options

Exhaustive exploration is an automatic state space exploration algorithm intended for small SDL systems where the requirements on correctness are very high.

The algorithm is a depth-first search through the state space similar to the bit state search, but there is no collision risk involved. The reason is that all traversed system states are stored in primary memory, so it is always possible to determine whether a newly generated system state has already been visited during the search.

The drawback with the algorithm is that very much primary memory is needed to be able to store all traversed states. This limits the complexity of the SDL systems the algorithm is applicable to.

Search Depth

The search depth is the maximum depth the Explorer will explore a particular execution path in the state space. When this depth is reached, the search is truncated and the search backs up to a previous system state.

- Default value: 100
- Command: Define-Exhaustive-Depth
- Menu choice: *Options2: Exhaustive: Depth*

MSC Verification Options

The MSC verification algorithm is a bit state exploration that is adapted to suit the needs of MSC verification:

- An MSC is always loaded to guide the search
- The search depth is different from the depth used during usual bit state exploration
- The search is aborted as soon as the MSC has been verified.

Search Depth

The maximum depth searched by the algorithm. The intention is that this depth always should be enough. If the MSC verification fails and the number of truncations is more than 0, this depth should be increased.

- Default value: 1,000
- Command: Define-MSC-Verification-Depth
- Menu choice: Not available

Timer Check Level

When verifying an MSC where there are timers in the MSC and/or in the SDL system, there is a choice of how to perform the matching between the timer events in the MSC and in the SDL system. The timer check level determines how this matching should be done:

- 0: No checking of timer events is performed.
- 1: If a timer event exists in the MSC a matching timer event must exist in the explored SDL path, but a timer event in the explored SDL path is accepted even if there is no corresponding MSC timer event.
- 2: All timer events in the MSC must match a corresponding timer event in the explored SDL path, and vice versa.

The choice must be determined by the style of MSC that is used.

This option affects the state space; see "Affecting the State Space" on page 2459.

- Default value: 1
- Command: Define-Timer-Check-Level
- Menu choice: *Options2: MSC: Timer check level*

Report Options

For each report type, you can define the action performed when the report is found and whether it should be reported to the user.

Report Action

The report action determines what action should be performed when a report situation is encountered while performing state space exploration. There are three possibilities:

- Continue: The search continues past the reported situation as if it never happened.
- Prune: The search is pruned and depending on the algorithm some appropriate action is taken. For example, when using bit state exploration, the search will back up one state and continue with the next alternative transition, as if max search depth was reached and the search truncated.
- Abort: The search is aborted and the command prompt displayed.

Note that for some report types, like Deadlock, the continue choice is impossible.

This option affects the state space; see "Affecting the State Space" on page 2459.

- Default value: Prune for all report types
- Commands: Define-Report-Continue, Define-Report-Prune and Define-Report-Abort
- Menu choices: *Options2: Report: Continue*, *Options2: Report: Prune* and *Options2: Report: Abort*

Report Log

The report log setting defines whether the report should be recorded in the list of generated reports. If the report log is set to Off for a particular report type, these reports will never show up in the report list. Note however that the report action still is performed, even though the report is not logged.

This option affects the state space; see "Affecting the State Space" on page 2459.

- Default value: On for all report types
- Command: Define-Report-Log
- Menu choice: *Options2: Report: Report log*

Report Viewer Autopopup

When an automatic state space exploration is finished, the Report Viewer is normally started automatically to present the found reports. In some cases this may be inconvenient, so there is a possibility to turn this feature off.

- Default value: On
- Command: Define-Report-Viewer-Autopopup
- Menu choice: *Options1: Report Viewer Auto Popup*

MSC Trace Options

When the Explorer performs an MSC trace, you can define what types of events that are traced.

Action Trace

By default, actions like tasks, decisions, etc. are not shown in the MSC trace. You may change this by setting action trace to On.

- Default value: Off
- Command: Define-MSC-Trace-Action
- Menu choice: Not available

State Trace

By default, changes in process states are shown in the MSC trace by adding a condition symbol. You may change this by setting state trace to Off.

- Default value: On
- Command: Define-MSC-Trace-State
- Menu choice: Not available

MSC Trace Autopopup

When you go to a report, an MSC Editor is normally started automatically to present the trace from the current root to the state where the report was generated. In some cases this may be inconvenient, so there is a possibility to turn this feature off.

- Default value: On
- Command: Define-MSC-Trace-Autopopup
- Menu choice: *Options1: MSC Trace Auto Popup*

State Space Options

The structure and size of the state space that can be generated for any given SDL system can be modified in a number of ways using the state space options. The default values are defined to make the state space as small as possible to make the Explorer immediately useful for as many applications as possible. This, however, also means that the search performed by the Explorer is fairly scarce compared to what is possible. Some error situations may thus be overlooked during the search if they only occur in a part of the state space that never is reached.

Since these options affect the state space, note the information in "Affecting the State Space" on page 2459.

Transition Type

There are two alternatives possible for the type of a behavior tree transition during state space exploration:

- It can be equal to a complete SDL process graph transition (the value "SDL" in the command)
- It can be a part of such an SDL transition (the value "Symbol-Sequence" in the command).

If it is equal to an SDL process graph transition, whenever such a transition is started, it is completed before anything else is allowed to happen. This implies that all process instances in all system states in the behavior tree will always be in an SDL process graph state.

If it is only a part of an SDL process graph transition, a transition in the behavior tree is considered to be a sequence of events that are local to the process instance, followed by a non-local event. Examples of local events are tasks and decisions; examples of non-local events are creates

and inputs/outputs of signals from/to other process instances. The idea of this alternative is to model the ITU semantics for SDL as closely as possible while still allowing optimized performance during state space exploration.

- Default value: SDL
- Command: Define-Transition
- Menu choice: *Options1: State Space: Transition*

Scheduling Algorithm

The scheduling algorithm defines which of the process instances in a system state will be allowed to execute. There are two possible alternatives:

- All of the process instances in the ready queue are allowed to execute (the value “All” in the command)
- Only the first process instance in the ready queue is allowed to execute (the value “First” in the command).

The ready queue is a queue containing all process instances that have received a signal that can cause an immediate transition, but that have not yet had the opportunity to execute this transition to its end.

If all process instances are allowed to execute, the semantics of ITU recommendation Z.100 are modeled. There will be one child node to the current node in the behavior tree for each process instance in the ready queue.

If only the first process instance is allowed to execute, the semantics of an application that has been generated by the SDL to C Compiler are modeled. There will only be one child node to the current node in the behavior tree, the first process instance in the ready queue.

- Default value: First
- Command: Define-Scheduling
- Menu choice: *Options1: State Space: Scheduling*

Event Priorities

The events that are represented in a behavior tree can be divided into five classes:

- Internal events: Events local to the processes in the system, e.g., tasks, decisions, inputs, outputs.
- Input from ENV: Reception of signals from the environment. The signal is put in the input port of a process instance or on a channel queue.
- Timeout events: Expiration of SDL timers. The timer signal is put in the input port of a process instance.
- Channel outputs: A signal is removed from a channel queue and put into another channel queue or the input port of a process instance
- Spontaneous transitions: A transition in a process caused by input of `none`.

To each of these event classes a priority of 1, 2, 3, 4 or 5 is assigned. These priorities are used during state space exploration to determine which transitions should be generated from each system state. The events with priority 1 are first considered. Only if no events with priority 1 are possible in the current state, the events with priority 2 are considered. Only if no events with priority 1 or 2 are possible in the current state are events with priority 3 considered, etc.

Note that also the setting of the symbol time option will have an impact on the events that will can be executed in each system state; see section "Transition Time" on page 2469.

The two most common ways of assigning priorities to event classes are:

- All event classes are assigned priority 1.
- Internal events and channel outputs are assigned priority 1, and external, timeout and spontaneous transition events are assigned priority 2 (the default).

The first alternative represents the situation where no assumptions can be made about the time scale for the different types of events. The second alternative represents a situation where the internal delays are very short compared to the timeout durations and execution speed of the environment.

- Default value: Priorities 1, 2, 2, 1, 2
- Command: Define-Priorities
- Menu choice: *Options1: State Space: Priorities*

Transition Time

A common simplification made in the analysis of SDL systems is to consider the time it takes for a process to execute a symbol, e.g. an action or output, to be zero. This time is of course never zero in a real system, but in many cases the time is very small compared to the timer durations in the system, and can be neglected when analyzing the system.

Consider for example a situation where a process sets a timer with a duration 5 and then executes something that may take a long time, e.g. a long loop, and then sets a timer with duration 1. If symbol time is assumed to be zero, the second timer will always expire first. If considered to be non-zero, any one of the timers can potentially expire first.

The explorer allows the user to choose whether to assume that the execution time for SDL symbols is zero or undefined using the Define-Symbol-Time command.

- Default value: Zero
- Command: Define-Symbol-Time
- Menu choice: *Options1: State Space: Symbol time*

Channel Queues

The Explorer allows queues to be attached to and removed from all channels in the SDL system. If a queue is added for a channel, it implies that when a signal is sent transported on this channel it will be put into the queue associated with the channel. Then there will be a separate transition in the state space that represents the forwarding of the signal to the receiver (or the next channel queue).

- Default: No channels have queues
- Command: Define-Channel-Queue
- Menu choice: *Options1: State Space: Channel queues*

Maximum Input Port Length

The length of the input port queues is not infinite in the Explorer, since in practice it is likely to be a design error if the queues grow forever. If the length of a queue exceeds the defined max length during state space exploration, a “MaxQueuelength” report is generated.

- Default value: 3
- Command: Define-Max-Input-Port-Length
- Menu choice: *Options1: State Space: Input port length*

Maximum Transition Length

To make it possible to detect infinite loops within a transition in the state space, the maximum number of SDL symbols allowed to be executed in one transition is defined. If this number is exceeded during state space exploration, a "MaxTransLen" report is generated.

- Default value: 1,000
- Command: Define-Max-Transition-Length
- Menu choice: *Options1: State Space: Transition length*

Maximum Number of Instances

To avoid infinite chains of create actions in the state space, the Explorer uses a max number of allowed process instances for any type. If this number is exceeded during state space exploration, a "Create" report is generated.

- Default value: 100
- Command: Define-Max-Instance
- Menu choice: *Options1: State Space: Max instance*

Maximum State Size

When the Explorer is exploring the state space, an internal buffer is used to store the system states. The size of this buffer defines the maximum size of the system states that the Explorer can handle.

- Default value: 100,000 (bytes)
- Command: Define-Max-State-Size
- Menu choice: *Options1: State Space: Max state size*

Timer Progress

One test that can be made with the Explorer is to look for non-progress loops, i.e. loops in the state space without any progress being made. The intention with this test is to look for situations where the SDL system is busy doing internal communication but to an outside observer looks dead.

This option defines if the expiration of a timer is considered as progress when performing non-progress loop checking. See also "Non Progress Loop Error" on page 2372 in chapter 52, *The SDL Explorer*.

- Default: On (timer expiration is considered to be progress)
- Command: Define-Timer-Progress
- Menu choice: *Options1: State Space: Timer progress*

Spontaneous Transition Progress

One test that can be made with the Explorer is to look for non-progress loops, i.e. loops in the state space without any progress being made. The intention with this test is to look for situations where the SDL system is busy doing internal communication but to an outside observer looks dead.

This option defines if a spontaneous transition is considered as progress when performing non-progress loop checking. See also "Non Progress Loop Error" on page 2372 in chapter 52, *The SDL Explorer*.

- Default: On (spontaneous transition is considered to be progress)
- Command: Define-Spontaneous-Transition-Progress
- Menu choice: Not available

Autolink Options

See section "Computing Test Cases" on page 1450 in chapter 35, *TTCN Test Suite Generation* for a discussion of the Autolink options.

Setting Advanced Options

Advanced options can be set for state space explorations to achieve a much larger state space than the default, thus allowing for special kind of errors to be detected. See "Using Advanced Validation" on page 2421 for more information.

To set advanced options, click the *Advanced* button in the *Explore* module. This executes the following set of commands:

```
Define-Scheduling All
Define-Priorities 1 1 1 1 1
Define-Max-Input-Port-Length 2
Define-Report-Log MaxQueuelength Off
```

The reasoning behind these settings are:

- The scheduling should be set to All, since we in this case are looking for signal races and a characteristic property of signal race conditions is that they are depending on the ordering of internal events.

- The priorities should be set to 1 for all types of events.
- To reduce the size of the state space, the maximum queue length should be set to a very small number. The reason is that when the environment is allowed to send signals to the system at any time, the queues that can receive signals from the environment will grow very rapidly.
- Since a lot of maximum queue length reports will be generated with these options, the report log for this report should be set to Off. Note also that the report action for this report should be Prune (which is the default).

References

[17] Holzmann, G.J:
Design and Validation of Computer Protocols
Prentice-Hall, 1991
ISBN 0-13-539834-7

Chapter

54 *The SDL Analyzer*

The SDL Analyzer's primary task is to check SDL-92 diagrams (extended as defined in "Compatibility with ITU SDL" on page 2 in chapter 1, *Compatibility Notes, in the Release Guide*) with respect to syntactic and semantic rules. The Analyzer accepts both SDL/GR and SDL/PR as input, and has the ability to transform SDL/GR diagrams to SDL/PR files and vice-versa.

This chapter is a reference to the Analyzer commands. Its limitations according to the Z.100 recommendation is found in "SDL Analyzer" on page 20 in chapter 1, *Compatibility Notes, in the Release Guide*.

For a guide to how to perform syntactic and semantic check on an SDL diagram structure, see chapter 55, *Analyzing a System*. This chapter also gives an overview of the Analyzer.

The Analyzer User Interfaces

The Analyzer Graphical UI

The Analyzer's graphical user interface is managed by the Organizer.

To start the Analyzer, select *Analyze* from the Organizer *Generate* menu or click the *Analyze* quick button. For more information, see "Analyze" on page 112 in chapter 2, *The Organizer* and "Quick Buttons" on page 180 in chapter 2, *The Organizer*.

The Analyzer's graphical user interface is further described in "Using the Analyzer" on page 2618 in chapter 55, *Analyzing a System*.

The Analyzer Batch UI

See *"Batch Facilities" on page 208 in chapter 2, The Organizer* for information on running the Analyzer non-interactively.

The Analyzer Command Line UI

The Analyzer also has a *command line mode,* when it is started from the OS outside the graphical environment of the SDL Suite. This is the user interface that is described in the remainder of this chapter.

Starting the Analyzer

The Analyzer is started in command line mode from the OS prompt with the command:

```
sdtsan [1]
```

When the Analyzer is started in command line mode, it responds with the prompt

```
SDL Analyzer
Command :
```

1. The directory where the SDL Suite binaries are stored is assumed to be included in the PATH variable. Otherwise, the complete directory path must be supplied.

Environment Variables

- SDTSAN_WARN_GATE

  By setting this to any value the analyzer will give warnings instead of errors for unconnected gates that should be connected. This option is only provided for backwards compatibility and should not be used if possible.

- SDTSAN_ASN1_ENUM_OP

  By setting this to any value the analyzer will pass the -e option to asn1util. This will create all operators for ASN.1 ENUMERATED types as defined in Z.105.

- SDTSAN_FILTER_USE

  By setting this to any value the analyzer will run the specified filter command specified in the analyzer options once before SDL package dependencies are resolved. This makes it possible for external filter scripts to exclude SDL package before make files are generated.

- SDTSAN_FILTER_ABORT

  By setting this to any value the analyzer will terminate it’s job at the first failing filter command.

- SDTSAN_INCLUDE_PATH

  Set this to a list of directories where the analyzer should look for include files. The separator is : on Unix and ; on Windows.

Syntax of Analyzer Commands

A command name may be abbreviated by giving sufficient characters to distinguish it from other command names. A hyphen (’-’) is used to separate command names into distinct parts.

Any part may be abbreviated as long as the command does not become ambiguous.

Parameters to commands are separated by one or several spaces. In case you omit a parameter of type “on/off”, the value currently defined will toggle between its “on” and “off” values.

In both command names and parameters there is no distinction made between upper and lower case letters.

> **Note:**
>
> **On UNIX** systems, distinction is made between upper case letters and lower case letters for file names.

Comments are indicated by the "#" character, and extend to the end of the line. Comments may only be preceded by white space.

Description of Analyzer Commands

In this section, the Analyzer commands are listed in alphabetical order, along with their parameters and a description.

! (shell escape)

Parameters:

```
<operating system command>
```

Escape to the shell to do command.

Add-Input

Parameters:

```
<Filespec>
```

Add a file to the list of files to be converted to PR. The file will be processed according to the setting of Input-Mode.

Analyze

Parameters:

```
(None)
```

This command orders the Analyzer to perform the passes according to the *Analyzer options* as they are defined.

> **Hint:**
>
> The current analysis options can be printed using the command *Show-Analyze-Options* (see "Show-Analyze-Options" on page 2497).

The order in which the passes are performed is:

1. *Conversion to PR*, if the input is C header, ASN.1 or SDL/GR diagrams.
2. *Macro expansion* (optional).
3. *Syntactic analysis* (optional).
4. *Pretty-printing* of a PR file (optional).
5. *PR to GR conversion* (optional). This pass converts an SDL/PR input file to SDL/GR diagrams, using the native format of the SDL Suite.
6. *Semantic analysis* (optional).
7. *Cross reference generation* (optional).

Note:

The Analyzer may need to be reset (using the *New* or *Clear* command) between subsequent *Analyze* commands, since the Analyzer keeps track of what passes have been performed in order to minimize analysis time.

ASN1-Coder-Name

Parameters:

```
<name>
```

The prefix of encode and decode functions for ASN.1 types. Default is BER.

ASN1-Keyword-File

Parameters:

```
<file>
```

Change keywords in files output by asn1util according to file. Default is no file, in this case the keywords in file asn1util_kwd.txt in the SDT installation is used.

Asn1Util

Parameters:

```
<options>
```

Invoke asn1util through the post master. Same options as the command line interface.

Cd

Parameters:

`<directory>`

Change current working directory to <directory>.

Clear

Parameters:

`(None)`

Force the Analyzer to perform all passes from scratch as defined in the Analyzer options on the previously selected input. See also the *New* command described in "New" on page 2484.

Coder-Buffer-In-Sdl

Parameters:

`Octet-String | Coder-Buf | Custom-Coder-Buf | None`

Pick representation of ASN.1 Buffer in SDL, None means no encoding available in SDL. Default is Octet-String. For more information see "Buffer Management System" on page 2850 in chapter 58, *ASN.1 En-coding and De-coding in the SDL Suite*.

ComplexityMeasurement-File

Parameters:

`[<file spec>]`

Specifies where the complexity measurements file is stored if the Set-ComplexityMeasurement command is set.

Default is `<system name>.csv`.

See chapter 48, *Complexity Measurements* for more information.

Component

Parameters:

`[<SDL qualifier>]`

Add a new item to a program. See *"Partitioning" on page 2641 in chapter 56, The Cadvanced/Cbasic SDL to C Compiler*.

Cpp2sdl

Parameters:

<options>

Invoke cpp2sdl through the post master. Same options as the command line interface.

Env-Header-Channel-Name

Parameters:

```
[<name>]
```

Configuration of channel names in environment header file (.ifc). %n is the name of the entity in SDL, %s is the name of the scope. Omit <name> to exclude the definitions completely from the file. Default is %n.

Env-Header-Literal-Name

Parameters:

```
[<name>]
```

Configuration of literal names in environment header file (.ifc). %n is the name of the entity in SDL, %s is the name of the scope. Omit <name> to exclude the definitions completely from the file. Default is %n.

Env-Header-Operators

Parameters:

```
[On/Off]
```

Configure if operators should be present in environment header file (.ifc). Default is On.

Env-Header-Signal-Name

Parameters:

```
[<name>]
```

Configuration of signal names in environment header file (.ifc). %n is the name of the entity in SDL, %s is the name of the scope. Omit <name> to exclude the definitions completely from the file. Default is %n.

Env-Header-Synonym-Name

Parameters:

```
[<name>]
```

Configuration of synonym names in environment header file (.ifc). %n is the name of the entity in SDL, %s is the name of the scope. Omit <name> to exclude the definitions completely from the file. Default is %n.

Env-Header-Type-Name

Parameters:

```
[<name>]
```

Configuration of type names in environment header file (.ifc). %n is the name of the entity in SDL, %s is the name of the scope. Type definitions are always present in the file. Default is %n.

Error-File

Parameters:

```
[<file spec>]
```

An optional file to store analyzer messages in. Default is no error file.

Exit

Parameters:

```
(None)
```

Terminates the Analyzer and returns you to the Operating System prompt.

Filter

Parameters:

```
[Command]
```

This command allows filtering or preprocessing files before they are read by the Analyzer. The command should be an executable file, preferably found in $PATH. The Analyzer appends three parameters to the supplied command. The first parameter is the name of the file, the second parameter tells what the Analyzer is about to do with the file, and the third is the Source Directory. Currently the following are available:

- **use** (before SDL package dependencies are resolved, this phase is only activated when the environment variable SDTSAN_FILTER_USE is set)
- **import** (before converting to SDL PR)
- **macro** (before macro expansion)
- **parse** (before syntax analysis)
- **include** (before syntax analysis)

The command is supposed to modify the file given in the parameter and leave the result in the same file. Note that the files passed are your original files.

Generate-Advanced-C

Parameters:

(None)

This command invokes the Cadvanced SDL to C Compiler, which will produce one / multiple C program(s), as well as a makefile. This C program is then compiled and linked in order to build up an executable application. The Cadvanced SDL to C Compiler uses the options which have been previously defined using the various Generate Options commands. See also Generate-Basic-C.

Generate-Basic-C

Parameters:

(None)

This command invokes the Cbasic SDL to C Compiler, which will produce one / multiple C program(s), as well as a makefile. This C program is then compiled and linked in order to build up an executable application. The Cbasic SDL to C Compiler uses the options which have been previously defined using the various Generate Options commands. This command cannot be used when building applications, but otherwise it is identical to Generate-Advanced-C.

Generate-Micro-C

Parameters:[1]

(None)

This command invokes the Cmicro SDL to C Compiler, which will produce one / multiple C program(s), as well as a makefile. This C program is then compiled and linked in order to build up an executable application. The Cmicro SDL to C Compiler uses the options which have been previously defined using the various Generate Options commands. See also Generate-Advanced-C.

GR-PR-File

Parameters:

[<file spec>]

This command specifies where the output of the conversion to PR pass is stored. Default is <infile>.pr.

1. The exact syntax is subject to change since the tool is still under development.

GR2PR

Parameters:

```
<GR-File> <PR-File>
```

The command converts the file designated by the file specifier `GR-File` to a textual (i.e. PR) description of the diagram, storing the results in the file designated by the file specifier `PR-File`.

The file `GR-File` should contain an SDL diagram stored in binary format.

Help

Parameters:

```
[Command]
```

The command issues on-line help for the Analyzer commands. Depending on if a command is specified or not, the following happens:

- By typing this command without parameters, the available commands are listed with their parameters. Control is then transferred to the Analyzer on-line help, where additional information about all commands is available. The on-line help prompt looks like this:

  ```
  Topic?
  ```

  – Typing a command name followed by `<Enter>` issues information about the command.

  – Typing `<Enter>` only, returns control to the Analyzer prompt.

- Typing the Help command followed by a command name, on-line for that command will be displayed. Control is then transferred to the Analyzer on-line help utility.

You can abbreviate any topic name, although ambiguous abbreviations result in all matches being displayed.

Include-Directory

Parameters:

```
[<directory>]
```

Where the analyzer looks for include files. An empty <directory> clears the list.Subsequent commands adds more directories to the list. Default is an empty list.

Include-File

Parameters:

```
<file spec>
```

Execute analyzer commands in <file spec>.

Include-Map

Parameters:

```
<logical name> <file spec>
```

Maps logical names to file names for graphical includes when converting graphical SDL (GR) to text (PR). This command should be preceded by commands to specify input mode to GR and a GR file. Subsequent commands adds more maps to the list. Default is an empty list.

Input-Mode

Parameters:

```
ASN.1 | C-Header | GR | PR
```

Select filter to convert input to PR. Default is PR.

Instance-File

Parameters:

```
[<file spec>]
```

Where the instance tree is stored. Default is `<system name>.ins`. See "SDL Instance Information" on page 2512 for more information.

Macro-PR-File

Parameters:

```
[<file spec>]
```

This command specifies where the output of the macro expander is stored. Default is `<infile>.prm`.

Make-File

Parameters:

```
[<file spec>]
```

Name of makefile. Default is `<system name>.m`.

Make-Template-File

Parameters:

```
[<file spec>]
```

Name of make template file. Default is no template.

New

Parameters:

```
(None)
```

This command initializes the Analyzer. Following the *New* command, you will need to specify an input to the Analyzer using the *Set-Input* command.

See also "Clear" on page 2478.

Operating-System

Parameters:

```
(None)
```

Creates a subprocess which returns you temporarily to the Operating System. Once you terminate this process, control is returned to the Analyzer.

Organizer-Object

Parameters:

```
Level Type Name File Sep SepName Selected [Depend]
```

Pass the Organizers info on a diagram to the Analyzer. This command is used by the Organizer.

Pretty-PR-File

Parameters:

```
<Filespec>
```

This command specifies where the output of the pretty printer is stored. Default is `<infile>.sdl`.

Program

Parameters:

```
[<name>]
```

Name of program built from components. See *"Partitioning" on page 2641 in chapter 56, The Cadvanced/Cbasic SDL to C Compiler*.

Quit

Parameters:

```
(None)
```

Synonym for exit.

SDL-Coder-Name

Parameters:

```
<name>
```

The prefix of encode and decode functions for SDL types. Default is ASCII.

SDL-Keyword-File

Parameters:

```
[<file spec>]
```

Produce a list of SDL keywords and SDT Ref in a file. Default is no list.

SDT-Ref

Parameters:

```
[<sdt ref>]
```

Use <sdt ref> in error messages from command parser overriding any other <sdt ref>.

SDT-SYSTEM-6.3

Parameters:

```
(None)
```

The Organizer uses this to make sure the right version of the analyzer is running.

Set-ASN1-Coder

Parameters:

```
[On/Off]
```

Generates encoder and decoder from ASN.1 modules (requires separate license).

Default is off.

Set-C-Compiler-Driver

Parameters:

```
[On/Off]
```

This option informs the SDL to C compilers if the SDL C Compiler Driver should be invoked. See chapter 60, *SDL C Compiler Driver (SC-CD)*.

Default is off.

Set-C-Plus-Plus

Parameters:

`[On/Off]`

SDL to C compilers generate code to be compiled with C++ compiler if on. Default is off.

Set-Case-Sensitive

Parameters:

`[On/Off]`

Case sensitive SDL names if On. Keywords must be all upper or lower case if on.

Default is off.

Set-Compile-Link

Parameters:

`[On/Off]`

This option informs the Code Generator (SDL to C compilers) whether the generated code should be compiled and linked automatically (by executing the generated makefile).

Default is on.

Set-ComplexityMeasurement

Parameters:

`[On/Off]`

This option will generate a complexity measurement file. Following the *Set-ComplexityMeasurement* command, you may specify a file name using the ComplexityMeasurement-File command.

Default is off.

See chapter 48, *Complexity Measurements* for more information.

Set-Echo

Parameters:

`[On/Off]`

This option will print (echo) all Analyzer commands as they are executed. Default is off.

Set-Env-Function

Parameters:

`[On/Off]`

This option informs the code generators (SDL to C compilers) if environment functions should be generated or not.

Default is off.

Set-Env-Header

Parameters:

`[On/Off]`

This option is applicable to the SDL to C compilers only. With this parameter turned on, a header file containing the definitions of the SDL system's interface to the environment is created. This file is identified by the `.ifc` file name extension.

Default is off.

Set-Error-Limit

Parameters:

`<Errorlimit>`

This command sets the limit of reported errors after which the Analyzer will stop its execution. Use 0 to turn off the limit.

Default is a limit of 30 diagnostics.

Set-Expand-Include

Parameters:

`[On/Off]`

This option specifies whether include directives should be handled and expanded when reading input files. See "Including PR Files" on page 2506.

Default is on.

Set-Expression-Limit

Parameters:

`<Depth>`

This command instructs the Analyzer to issue a warning when the semantic analyzer encounters an SDL expression which is at least *Depth* deep.

> **Note:**
>
> Deeply nested expressions may cause a **significant degradation of performance** when performing the semantic analysis pass. It is therefore recommended to break down complex expressions into multiple, less complex expressions.

Default is 0, meaning that no warnings will be issued.

Set-External-Type-Free-Function

Parameters:

```
[On/Off]
```

This option informs the code generators if GenericFree should be called for variables of externally defined types (of struct kind).

Default is on.

Set-File-Prefix

Parameters:

```
[<file name prefix>]
```

Prefix for files generated by the code generators. Default is no prefix.

Set-Full

Parameters:

```
[On/Off]
```

Redo everything if on. Default is off.

Set-Generate-All-Files

Parameters:

```
[On/Off]
```

Existing files will not be touched by C code generators if the new version is identical to the existing one and this option if Off. Default is On.

Set-Ignore-Hidden

Parameters:

```
[On/Off]
```

Decides if PR should be generated for hidden SDL symbols. Default is On.

Set-Implicit-Type-Conversion

Parameters:

`[On/Off]`

When this option is on, the Analyzer allows implicit type conversion of reference data types (generators Own, ORef, and Ref). Note that analyzing large expressions with this option on is slow. Default is off.

For more information, see "Implicit Type Conversions" on page 134 in chapter 3, *Using SDL Extensions, in the SDL Suite 6.2 Methodology Guidelines*.

Set-Input

Parameters:

`<Filespec>`

The Analyzer will use the file specified in filespec as input file. The file will be processed according to the setting of Input-Mode (see "Input-Mode" on page 2483). Use Add-Input to process a system stored in several files.

Set-Instance

Parameters:

`[On/Off]`

With this option on, a file containing the SDL instance tree will be generated. Default is off. See "SDL Instance Information" on page 2512 for more information.

Set-Kernel

Parameters:

`<Kernel>`

This option is applicable to the SDL to C compilers only. This option allows you to select the appropriate kernel with predefined runtime libraries. The makefile generated by the SDL to C compilers contains instructions which link the generated and compiled code with object modules of the selected library. The effect of this will be that the code which is generated by the SDL to C compilers will have different properties, reflecting the purpose of its usage (simulation, application generation, etc.). The allowed values of the parameter kernel vary from one configuration to another. The definitions are made in the file `sdtsct.knl`. The definitions for Cadvanced are made in the file `sdtsct.knl`, whereas the definitions for Cmicro are made in the file `scmsct.knl`.

The following values are recognized for Cadvanced:

Library	Application area
SCTADEBCOM	Simulation, simulated time
SCTADEBCLCOM	Simulation, real time
SCTAPERFSIM	Performance simulation
SCTAAPPLCLENV	Application
SCTADEBCLENV	Application debug (simulation with environment, only Cadvanced)
SCTAVALIDATOR	Validation
SCTATTCNLINK	TTCN link

The following values are recognized for Cmicro:

Library	Application area
SCMADEBCOM	SDL Target Tester and communication, no real time clock
SCMADEBCLCOM	SDL Target Tester, communication and real time clock
SCMADEBCLENVCOM	SDL Target Tester, communication, user's env and real time clock
SCMAAPPLCLENVMIN	Application with environment, clock, etc.
SCMAAPPLCLENV	Evaluation Kernel (no complex ADT; no SDL Target Tester)

Note:

The predefined kernels for Cmicro cannot be used with any C compiler. They require GCC 2.95.2 **on UNIX** or Microsoft Visual C++ 6.0 **in Windows**.

Set-Lower-Case

Parameters:

`[On/Off]`

SDL does not distinguish between lower case letters (a..z) and upper case letters (A..Z), but the high level programming language C does. With this parameter turned on, the names of, for instance, variables in the generated C code is written in lower case letters only. Otherwise, the case of an item in the SDL definition is used.

Default is off.

Set-Macro

Parameters:

`[On/Off]`

Turns on or off the macro expansion pass. This option must be turned on if your diagram contains references to macro(s).

Default is off.

Set-Make

Parameters:

`[On/Off]`

This command affects an option which enables or disables the generation of a makefile in conjunction with generation of C code.

Default is on.

Set-Modularity

Parameters:

`[Modularity]`

Allows to specify the modularity of the code generated by the code generators, e.g. the SDL to C compilers. This option affects how many files will be generated.

Three options for the parameter *Modularity* are available:

- *No* – One single code module is generated.
- *User* – User defined, according to the definitions in the Organizer. See "Edit Separation" on page 137 in chapter 2, *The Organizer*.
- *Full* – Multiple code modules will be generated.

Default value is *No*.

Set-Optional-Make-Operator

Parameters:

```
[On/Off]
```

Optional and default struct parameters are included in the make operator (invoked by (. .)) if On. Default is Off.

Set-Output

Parameters:

```
<Filespec>
```

Allows you to save the information generated by the Analyzer in files with a different name, using *Filespec* as the base name.

File name extensions are appended automatically by the Analyzer.

Set-Pr2Gr

Parameters:

```
[On/Off]
```

Turning this option on, the Analyzer will perform the PR to GR conversion pass. The steps are:

1. The Analyzer creates diagram files with temporary names.
2. The Analyzer lets the SDL Editor do tidy up on the diagram files.
3. The Organizer renames/moves diagram files according to its preference.

Step 3 is only performed when a PR to GR conversion is run from the Organizer. The request to put temporary files in the current working directory is ignored on the Windows platform (they end up in some temp directory).

Default is off.

Set-Predefined-XRef

Parameters:

```
[On/Off]
```

Include predefined data in SDL cross reference file if On. Default is Off.

Set-Prefix

Parameters:

`[Prefix]`

Allows you to specify what prefix the code generators add to SDL names in the generated code.

Possible values for the parameter *prefix* are:

- *No*
- *Entity*
- *Full*
- *Special*

Default is *Full*.

Set-Pretty

Parameters:

`[On/Off]`

Turning this switch on, a pretty printed PR file will be generated.

Default is off.

Set-References

Parameters:

`[On/Off]`

This option allows you to disable the verification that exactly one reference corresponds to each remote definition.

Default is on.

Set-SDL-Coder

Parameters:

`[On/Off]`

Generate encoder and decoder from SDL (requires separate license).

Default is off.

Set-Sdt-Ref

Parameters:

`[On/Off]`

Decides if SDT references should be included in the generated code as comments for easier navigation to its source. As default, SDT references in comments are included. Note that omitting SDT references in

comments by using this command is not supported by standard IBM Rational code generators.

Set-Semantic

Parameters:

`[On/Off]`

With this option turned on, the semantic check pass will be performed. Otherwise, the Analyzer will stop after the syntactic check.

Default is on.

See also "Optimizing a System to Reduce Analysis Time" on page 2621 in chapter 55, *Analyzing a System*.

Set-Signal-Number

Parameters:

`[On/Off]`

Code generators will generate a file with signal numbers if on.

Default is off

Set-Source

Parameters:

`[On/Off]`

This command informs the code generators whether to generate C code or not when a generate code command is given. The reason for disabling code generation is that you may want to produce a makefile only, without generating code.

Default is on.

Set-Synonym

Parameters:

`[On/Off]`

Decides if the specified synonym file should be used or not. Default is Off.

Set-Syntax

Parameters:

`[On/Off]`

This option turns on or off the syntax analysis pass.

Default is on.

Set-TAEX-Make

Parameters:

`[On/Off]`

This command controls the generation of a makefile for Targeting Expert in conjunction to generation of C code.

Default is off.

Set-Upcase-Keyword-Pretty

Parameters:

`[On/Off]`

Select between upper or lower case keywords in pretty printings.

Default is off (i.e. lower case letters).

Set-Warn-Else-Answer

Parameters:

`[On/Off]`

Print a message if a decision or transition option does not have an else answer and the Analyzer is unable to decide that it is not needed.

Default is on.

Set-Warn-Match-Answer

Parameters:

`[On/Off]`

Print a message if a decision or transition option does not have an answer for some value. The analyzer is not able to do this in all situations.

Default is on.

Set-Warn-Parameter-Mismatch

Parameters:

`[On/Off]`

Print a message if the types of the actual and formal parameters are implemented with different typedef, i.e. they may be of different size. This is only checked for functions that cannot handle length mismatch. No warning is given if the actual parameter is a literal.

Default is on.

Set-Warn-Optional-Parameter

Parameters:

`[On/Off]`

Print a message if an optional actual parameter is omitted.

Default is on.

Set-Warn-Output

Parameters:

`[On/Off]`

Print a warning message if an SDL signal output has a different semantic meaning between SDL 88 and 92.

Default is on.

> **Note:**
>
> This command is particularly useful when working with SDT 2.X diagrams in an SDT 3.X environment (SDT 2.X supports SDL-88, while SDT 3.X supports SDL-92).

Set-Warn-Parameter-Count

Parameters:

`[On/Off]`

Print a message if trailing actual parameters are omitted.

Default is on.

Set-Warn-Usage

Parameters:

`[On/Off]`

Print a message if an SDL definition is not used.

Default is off.

Set-Xref

Parameters:

`[On/Off]`

With this option on, a file containing *SDL Cross References* will be generated. See "SDL Cross-References" on page 2508.

Default is off.

Show-Analyze-Options

Parameters:

```
(None)
```

This command lists the current Analyzer options.

Show-Commands

Parameters:

```
(None)
```

Similar to *Help*. The commands are listed without the command parameters.

Show-Generate-Options

Parameters:

```
(None)
```

This command displays the code generator options as currently defined.

Show-License

Parameters:

```
(None)
```

This command returns information about:

- The total amount of software licenses
- How many licenses are currently in use, together with the user identification, the host name and the checkout time
- How many licenses are available

Show-Version

Parameters:

```
(None)
```

This command displays the version number of the Analyzer.

Source-Directory

Parameters:

```
[<directory>]
```

The analyzer looks for include files in this directory.

Default is current working directory.

Synonym-File

Parameters:

```
[<file spec>]
```

Specifies the synonym file (containing values of external synonyms). Default is no synonym file.

TAEX-Make-File

Parameters:

```
[<file spec>]
```

Name of makefile for Targeting Expert. See *Set-TAEX-Make*.

Target-Directory

Parameters:

```
[<directory>]
```

Specifies the directory where the code generators puts their files.

Default is current working directory.

Thread

Parameters:

```
<name> <stacksize> <stackprio> <maxqueuesize>
<maxmessize>
```

Use this command to distribute components of a program over several threads. Components following this command belongs to this thread. Parameters at the end may be omitted. Omitted parameters will use default values. The word default can be used to indicate that the default value is desired for a parameter. For more information see "Threaded Integration" on page 3303 in chapter 64, *Integration with Operating Systems*.

XRef-File

Parameters:

```
[<file spec>]
```

Specifies where the cross references are stored.

Default is `<infile>.xrf`.

Miscellaneous Analyzer Commands

The following commands can be used for instance in the case of malfunction or unexpected behavior. They are not described further within the scope of the user documentation.

zzMake-Options
Parameters:
(None)

zzSet-Access-Path-Tab
Parameters:
(None)

zzSet-Optimize
Parameters:
(None)

zzSet-Organizer-File
Parameters:
(None)

zzSet-Over-View
Parameters:
(None)

zzSet-SDT
Parameters:
(None)

zzSet-Verbose
Parameters:
(None)

zzSet-Warn-Oper-Usage
Parameters:
(None)

zzShow-Error
Parameters:
(None)

zzShow-Organizer
Parameters:
(None)

zzTransport
Parameters:
(None)

zzWrite-Name-List
Parameters:
(None)

zzWrite-Path
Parameters:
(None)

zzWrite-Pretty
Parameters:
(None)

zzWrite-Symbol
Parameters:
(None)

zzWrite-Syntax
Parameters:
(None)

Conversion to PR

During the first pass of the Analyzer, an SDL/PR file suitable for processing by the following passes (Macro Expansion or Syntax analysis) is produced. The resulting PR file is not intended to be read by the human, and is not formatted for that purpose. Conversion to PR is needed when input is not already in SDL/PR format, i.e, for SDL/GR diagrams, ASN.1 files, and C header files.

Several tools might be invoked through the PostMaster to perform the actual conversion:

- The SDL Editor to process SDL/GR. See "GR to PR Conversion" on page 2047 in chapter 43, *Using the SDL Editor*.
- The ASN.1 Utilities to process ASN.1. See "Translation of ASN.1 to SDL" on page 704 in chapter 13, *The ASN.1 Utilities*.
- The CPP2SDL Utility to process C/C++ header files. See "C/C++ to SDL Translation Rules" on page 785 in chapter 14, *The CPP2SDL Tool*.

Additional SDL/PR files may be copied into the output of this pass.

A number of checks are performed during this conversion pass. Many of these are actually syntactic in their nature. They may result in error messages; see "Error and Warning Messages" on page 2531.

The Macro Expander

A macro is a form of text substitution used in the SDL system definitions. In a macro definition, a collection of lexical units is defined. A macro call indicates where the text from the macro definition body is to be included. All macro calls must be expanded before further analysis of the system definition can be done.

> **Note:**
>
> **Macros may not contain any states.**
>
> **Diagrams referenced in PR form and macros only called in PR will not be converted. If a macro is called only in PR form, the macro definition must be in PR and placed in a diagram that will be converted.**

The name of the macro definition is visible in the whole system definition. A macro call may appear before its corresponding macro definition. A macro definition may contain macro calls, but not calls to itself, directly or indirectly.

A macro may use parameters, called formal parameters, in macro definitions and actual parameters in macro calls. There must be an equal number of formal parameters and actual parameters in corresponding macro definitions and macro calls. Using multiple formal parameters with the same name is not allowed.

When a character string is used as an actual parameter, the value of the character string will be used to replace the formal parameter in the macro body and not the character string itself.

The keyword *MACROID* may be used in a macro definition body only. The *MACROID* keyword will be replaced by a unique name at each expansion of the macro (that is, only one unique name is available at each expansion of a macro definition). New names can be constructed by concatenating names, parameters and *MACROID* using a percent sign (%) as a separator.

The reserved word *MACRODEFINITION* is not allowed inside a macro definition. The reserved words *MACROID* and *ENDMACRO* are only allowed inside a macro definition.

Macro expansion follows this order:

1. The name in the macro call is used to find the corresponding macro definition, and a copy of that macro definition is made.
2. In this copy, all occurrences of formal parameters are replaced by actual parameters. All occurrences of *MACROID* are replaced by a unique name. Also, all percent signs (%) separating names are removed.
3. Macro calls in the modified macro definition body are expanded.
4. The modified and expanded text is substituted for the macro call text in the system definition.

Implementation Details

The syntax checks made in the macro parser are entirely specific to macros since no other checks are possible until all macro calls are expanded. Some semantic checks are made during the expansion. These may produce error messages; see "Error and Warning Messages" on page 2531.

Example 346: Macro in SDL that Will Not Work

Constructs such as:

```
macro test('if par3 then',
           'if not par3 then',
           'k<0') ;
```

where the text `par3` in the first and second parameter strings is intended to be replaced by the third formal parameter in the macro definition will not work. This is because all formal parameters are replaced with actual parameters in a single step.

The reserved word *MACROID* is replaced by a unique name consisting of the string *"XMID"* and an integer. To assure system wide uniqueness the name is tested against all existing names in the system and the integer will be increased until the name is unique.

The Lexical and Syntactic Analyzer

The scanner (*lexical analyzer*) reads the input file and passes on lexical units to the parser. The parser checks the syntax and builds an internal representation (an abstract syntax tree) for the SDL specification.

The lexical and syntactic analysis may be performed not only on complete system definitions, but also on the following SDL units:

- *Block*
- *Process*
- *Substructure*
- *Service*
- *Procedure*
- *Package*

If no errors are found during the syntactic analysis, the syntax tree can be passed to the pretty-printer and, if the input is a system or package definition, to the semantic analyzer. For a description of the errors that can produced, see "Error and Warning Messages" on page 2531.

Separate Analysis

You may perform analysis on separate SDL units. The units that can be handled (other than system) are:

- *Block*
- *Process*
- *Substructure*
- *Service*
- *Procedure*
- *Package*

> **Note:**
>
> When performing separate analysis on a package, referred packages must be supplied as well.

If a unit is given without context, only syntactic checking can be performed. But, if the unit's placement in the system (that is, its context) is also given, both the syntactic checking and most of the semantic checking can be performed. The context of a unit is its enclosing scope units including the definitions that are not referenced, i.e. remote diagrams are excluded.

GR Input

In the Analyzer, the context is specified in the Organizer tree structure if the input is a GR diagram. The diagrams surrounding the diagram that is to be separately analyzed and the diagrams inside that diagram are translated to PR and analyzed. No errors will occur due to missing remote diagrams, except missing remote diagrams inside the separately analyzed diagram. See "Performing Syntax Check" on page 2620 in chapter 55, *Analyzing a System*.

PR Input

If the input is a PR file, there is no way to specify what part of the system to Analyze. Only syntax analysis is available on incomplete systems in PR form. See "Converting SDL/PR to SDL/GR" on page 2628 in chapter 55, *Analyzing a System*.

Including PR Files

The Analyzer allows the user to divide the SDL description into a number of separate PR files. (The macro expansion can, however, only handle one file at a time.) It may, for example, be convenient to have the system level data type definitions in a separate file. A separate file is included by adding an *include directive* to the SDL description.

Syntax of #INCLUDE Directives

The include directive is `#INCLUDE` followed by the name of the file which should be surrounded by single quotation marks. The directive should be placed in an SDL comment, directly after the comment start ("/*"), at the place where the file should be included.

- The first single quote must be preceded by at least one space. `<TAB>` characters are however not allowed.
- The second single quote may be followed by any number of spaces. `<TAB>` characters are however not allowed.

Example 347: #INCLUDE in Analyzer ----------

```
System Example;
/*#INCLUDE 'DataDefs.pr' */
/*#INCLUDE 'BlockA.pr' */
EndSystem Example;
```

Search Order for Included PR Files

The Analyzer will search for included SDL/PR files in the following order:

1. Any directory specified with Include-Directory command
2. Source directory
3. The current default directory
4. The directory designated by the environment variable `HOME`.
5. The directory designated by `$telelogic/sdt/include/ADT` **(on UNIX)**, or <*installation directory*>`\sdt\include\adt` **(in Windows)**
   (This directory contains a number of useful abstract data types in

SDL/PR that are included in the release. See chapter 62, *The ADT Library* for more information.)

The PR to GR Converter

General

The PR to GR Converter performs a conversion from SDL/PR files into SDL/GR diagrams. (It is also possible to input SDL/GR diagrams, which transforms them to new SDL/GR diagrams, applying default layouting algorithms.)

Conversion is done on a one-diagram-per-file basis, meaning that each generated GR diagram will be stored on one file. However, entering a PR file may generate multiple GR diagrams, depending on the contents of the PR file.

> **Note:**
>
> The PR to GR converter requires **syntactically correct** diagrams or PR files as input in order to function properly.
>
> Diagrams containing semantic errors will be converted, but, the **resulting** SDL/GR interpretation may be different from the SDL/PR representation.

Conversion Principles

The Analyzer passes which are performed are:

1. Macro expansion (if the PR input contains macro definitions).
2. Syntax analysis.
3. PR to GR conversion.
4. Then, graphical layouting is done by the SDL Editor.

Resulting files

The result from the PR to GR conversion is a number of SDL/GR binary files. Each diagram will be stored on one file.

To reconstruct the SDL diagram structure, the Organizer's command *Import SDL* should be used (see "Import SDL" on page 80 in chapter 2, *The Organizer*).

SDL Cross-References

The Analyzer has the ability to produce listings of SDL entities and references to these definitions. In one file, all *definitions* of entities in an SDL system will be gathered, along with cross references to these entities. Entities mainly used to indicate structure (such as system, block, substructure, process, procedure, and service), as well as entities defining objects (such as signals, variables and sorts) will be considered as definitions.

The file, which is described further below, is a plain text file.

> **Note:**
>
> The SDL Suite has support for graphical presentation of a cross-reference file, using an SDL-like graphical notation. To achieve this, you can use the Index Viewer. See chapter 46, *The SDL Index Viewer*.

Definitions and Cross References Files

The file name will be `<unitname>.xrf` for the *cross references file*, where <unitname> is the name of the SDL unit selected for analysis. Alternatively, the file name is to be supplied by the user when running the Analyzer from the Organizer.

If an SDL system is selected for analysis, all definitions and cross references are generated; while if case of a non-SDL system unit, definitions and cross references in the following units are generated:

- The unit itself
- All subunits to the selected unit
- All enclosing units (blocks and the system) of the selected unit

The following entities will be part of the files:

ACTIVE	ATLEAST

BLOCK	BLOCK SUBSTRUCTURE
CHANNEL	CHANNEL SUBSTRUCTURE
CONNECTION	CONTINUOUS SIGNAL
CREATE	DCL
DECISION	ENABLING CONDITION
EXPORT	EXPORTED_PROCEDURE
FORMAL PARAMETER	FPAR
GATE	GENERATOR
IMPORT	IMPORTED
IMPORTED VARIABLE	IN-CONNECTOR
INHERITS	INPUT
INSTANTIATION	JOIN
LITERAL	NEWTYPE
NEXTSTATE	NUMBER OF INSTANCES
OPERATOR	OUT-CONNECTOR
OUTPUT	PACKAGE_INTERFACE
PROCEDURE	PROCEDURE CALL
PROCEDURE PARAMETERS	PROCESS
PROCESS PARAMETERS	REMOTE PROCEDURE
REMOTE VARIABLE	RESET
SAVE	SERVICE
SET	SIGNAL
SIGNAL ROUTE	SIGNALLIST
SIGNALROUTE	SIGNALSET
SORT	STATE
STATE_LIST	SYNONYM
SYNTYPE	SYSTEM

TASK	TIMER
TRANSITION OPTION	USE
VARIABLE	VIEW
VIEWED	

Note:

No cross references will be generated for the predefined data types (INTEGER, NATURAL, CHARACTER, CHARSTRING, BOOLEAN, REAL, TIME, DURATION, PID) or for the predefined generators (ARRAY, STRING, POWERSET) as well as LITERALS and OPERATORS that are not part of the list above.

Syntax of Files

For each definition of any kind mentioned above the following information is generated:

```
ScopeLevel EntityType EntityName Reference
```

Explanation

- The number first on the line is the *scope level*. The system has level 1, any definition at the system level has scope level 2, and so on.
- Next on the line the *entity type* and *entity name* are given.
- Last there is an *SDT reference* to the place where the entity is defined. For more information on the format, see "Syntax" on page 917 in chapter 18, *SDT References*.

For each entity used to indicate structure (system, block, substructure, process, procedure, service) there is a header consisting of three additional lines, which should be considered as a comment to increase the readability.

Example 348: Reference in Analyzer Error

```
2  SIGNAL Bump \
SDTREF(SDL,/usr/tom/demongame.ssy(1),122(40,30),3)
```

Order of Definitions

The definitions are generated in pre-order (prefix walk in the tree formed by the definitions). This means that an entity is always followed by the entities defined within the entity. It also means that an entity at level N is defined in the first entity at level N-1 found when scanning upwards in the file.

Cross References

For each place where an entity is used, information about that cross reference is generated:

Example 349: Cross References ––––––––––––––––––––––––––––––

```
4  DCL Count \
#SDTREF(SDL,/usr/tom/game.spr(1),179(80,10),2)
     TASK \
#SDTREF(SDL,/usr/tom/game.spr(1),137(30,40),1)
     OUTPUT \
#SDTREF(SDL,/usr/tom/game.spr(1),128(80,55),2)
```

––

Cross references consist of a keyword, describing where the reference was found, and the SDL reference. In Example 349 on page 2511, the variable Count is referenced in a task symbol and in an output. All cross references for an entity are placed immediately after the line for the definition.

Note:

Entities used to indicate structure can also have cross references (procedure call, process create).

SDL Instance Information

The Analyzer supports the generation of a so-called instance information file, with the file extension `.ins`. This file contains various kinds of information about an SDL system. Similar to a cross reference file, it is a plain text file. The file syntax is described in detail in "File Syntax" on page 2513. The Analyzer invokes a tool called the Instance Generator to produce the instance information out of an analyzed SDL system.

The information that can be found in an instance information file helps to answer many questions. For example:

- How does a complex SDL-92 system look after instantiation of all types, i.e. when all inheritance hierarchies have been flattened out? Finding this information about the instance structure of a system directly from the SDL/PR file is a non-trivial task. Therefore, one of the main objectives behind the Instance Generator is to provide this "instance view" of a system.
- Which signals can be conveyed on a certain channel or signal route in a certain direction?
- What is the valid input signal set of a process?
- Which states does a process contain?
- Which transitions will take place if a certain signal is received by a process being in a certain state?
- Which output signals might be sent during a transition?
- Which procedures might get called during a transition?
- What is the set of possible nextstates after a transition?
- How shall a procedure be interpreted when called from a certain process?

The Instance Generator

The Analyzer provides two commands to use the Instance Generator from the command line user interface. These commands are described in "The Analyzer Command Line UI" on page 2474 but are repeated here for the sake of completeness.

Set-Instance

Parameters:

`[On/Off]`

This command sets an option that specifies whether an instance information file should be generated or not when the SDL system is analyzed. The option is by default off.

Instance-File

Parameters:

`<file spec>`

This command sets the name of the instance information file. The default filename is `<systemname>.ins`, where `<systemname>` is the name of the analyzed SDL system.

The Instance Generator can also be started from the Analyzer's graphical user interface. This is done in a way similar to how cross reference files are generated. In the Analyze dialog in the Organizer, you can check a button to make the Analyzer call the Instance Generator when the system has been analyzed. From this dialog it is also possible to set the name of the generated instance information file.

Note:

The Instance Generator is automatically invoked to produce the information needed by tools such as the State Matrix Viewer.

File Syntax

An instance information file consists of a *fileheader* followed by a list of *records*:

```
<fileheader>
<record>
<record>
...
<record>
```

The `<fileheader>` is a string telling which version of the Instance Generator that has generated the file. Each `<record>` describes an SDL entity according to the following format:

```
<level> <entity class> <name>
        <attribute> = <value>
        ...
        <attribute> = <value>
        .
```

The <level> is a number that tells at what depth in the "instance tree" the entity was found. For example, the system entity has level 1, a block entity in the system has level 2, a process entity in the block has level 3, and so on.

Attributes can be divided into three distinct **categories** depending on the format of their values:

1. A single value:

```
<attribute> = <single value>
```

2. A list of single values:

```
<attribute> = (
  <single value>
  <single value>
  ...
  <single value>
)
```

3. A list of pairs of single values:

```
<attribute> = (
  <single value> <single value
  <single value> <single value>
  ...
  <single value> <single value>
)
```

Sometimes an attribute is omitted from a record. This happens when the value of the attribute is an empty string or list.

The table below lists all entity classes for which information is generated in an instance information file. The attributes of the entities are also described, together with the category of the attributes (1, 2 or 3 according to above).

In the descriptions, the following terms are used:

- A *state-containing entity* is an entity that may contain states, i.e. a process, process instance, service, service instance or procedure.
- A *transition initiator* is an entity that can make a state-containing entity perform a transition from one state to another, i.e. a start, input, priority input, enabling condition or continuous signal.

Entity class	Attribute	Category	Description
BLOCK	InstRef	1	An SDT reference to the definition of this block.
	ConnectionOut	2	The names of the channels that are on the outside of this block.
	ConnectionIn	2	The names of the channels or signal-routes that are on the inside of this block.
BLOCK_INST	InstRef	1	An SDT reference to the definition of this block instance.
	TypeRef	2	SDT references to block type definitions. The first reference is to the block type from which this block instance is instantiated, followed by references to its supertypes all the way to the block type on top of the inheritance hierarchy.
CALLED_ PROCEDURE	InstRef	1	An SDT reference to the definition of this called procedure.
CHANNEL	InstRef	1	An SDT reference to the definition of this channel.
	SignalSet	3	The signals that are conveyed on this channel in the direction from the From-entity to the To-entity. Each signal is described by its name followed by an SDT reference to the definition of the signal.
	SignalSetRev	3	As SignalSet but for the signals that are conveyed in the opposite direction.

Entity class	Attribute	Category	Description
	From	1	The name of the block or block instance from which this channel starts. If the channel starts from the environment surrounding the system, system instance, block or block instance where this channel is located, this attribute is “env”.
	FromVia	1	This attribute is specified when the From-entity is a block instance, and is then the name of the gate of that block instance to which this channel is connected.
	To	1	The name of the block or block instance to which this channel leads. If the channel leads to the environment surrounding the system, system instance, block or block instance where this channel is located, this attribute is “env”.
	ToVia	1	This attribute is specified when the To-entity is a block instance, and is then the name of the gate of that block instance to which this channel is connected.
CHANNEL_SUBSTRUCTURE	InstRef	1	An SDT reference to the definition of this channel substructure.
CONTINUOUS_SIGNAL[a]	InstRef	1	An SDT reference to the definition of this continuous signal.
	InType	1	The name of the state-containing entity in which this continuous signal is defined.

Entity class	Attribute	Category	Description
	Outputs	3	The output signals that might be sent in the transition that is initiated by this continuous signal. Each signal is described by its name followed by an SDT reference to one of the output symbols in the transition that contains that name.
	Calls	3	The procedures that might get called in the transition that is initiated by this continuous signal. Each procedure is described by its name followed by an SDT reference to one of the symbols in the transition that contains a call to a procedure with that name. This name is also the name of a CALLED_ PROCEDURE entity that describes how the procedure look from the calling state-containing entity's point of view. This entity is normally placed immediately after the state information of the state-containing entity. However, when the state-containing entity is a procedure within another state-containing entity, it might be that the procedure has been called from another place in this surrounding state-containing entity. Then the CALLED_ PROCEDURE entity is not placed here also, since it is identical to the first one.
	NextStates	3	The states that might become the next-state after the transition that is initiated by this continuous signal. Each state is described by a name followed by an SDT reference to one of the state symbols that contains that name.

Entity class	Attribute	Cate-gory	Description
ENABLING_CONDITION	InstRef	1	See the description for CONTINUOUS_SIGNAL.
	InType	1	See the description for CONTINUOUS_SIGNAL.
	Outputs	3	See the description for CONTINUOUS_SIGNAL.
	Calls	3	See the description for CONTINUOUS_SIGNAL.
	NextStates	3	See the description for CONTINUOUS_SIGNAL.
GATE	InstRef	1	An SDT reference to the definition of this gate.
	SignalSet	3	The signals that are conveyed through this gate in to the block instance, process instance or service instance to which the gate belongs. Each signal is described by its name followed by an SDT reference to the definition of the signal.
	SignalSetRev	3	As SignalSet but for the signals that are conveyed in the opposite direction.
INPUT	InstRef	1	See the description for CONTINUOUS_SIGNAL.
	InType	1	See the description for CONTINUOUS_SIGNAL.
	Outputs	3	See the description for CONTINUOUS_SIGNAL.
	Calls	3	See the description for CONTINUOUS_SIGNAL.
	NextStates	3	See the description for CONTINUOUS_SIGNAL.

Entity class	Attribute	Category	Description
PRIORITY_INPUT	InstRef	1	See the description for CONTINUOUS_SIGNAL.
	InType	1	See the description for CONTINUOUS_SIGNAL.
	Outputs	3	See the description for CONTINUOUS_SIGNAL.
	Calls	3	See the description for CONTINUOUS_SIGNAL.
	NextStates	3	See the description for CONTINUOUS_SIGNAL.
PROCESS	InstRef	1	An SDT reference to the definition of this process.
	IniNoOfInst	1	The initial number of dynamic instances of this process.
	MaxNoOfInst	1	The maximum number of dynamic instances of this process. If this attribute is omitted it means that there is no upper limit on the number of dynamic instances.
	SignalSet	3	The signals that can be received by this process. This set of signals is commonly known as the valid input signal set of the process. Each signal is described by its name followed by an SDT reference to the definition of the signal.
	ConnectionOut	2	The names of the signalroutes that are on the outside of this process. Naturally, this attribute is only present when the process consists of services.

Entity class	Attribute	Category	Description
	ConnectionIn	2	The names of the signalroutes that are on the inside of this process. Naturally, this attribute is only present when the process consists of services.
PROCESS_INST	InstRef	1	An SDT reference to the definition of this process instance.
	TypeRef	2	SDT references to process type definitions. The first reference is to the process type from which this process instance is instantiated followed by references to its supertypes all the way to the process type on top of the inheritance hierarchy.
	IniNoOfInst	1	The initial number of dynamic instances of this process instance.
	MaxNoOfInst	1	The maximum number of dynamic instances of this process instance. If this attribute is omitted it means that there is no upper limit on the number of dynamic instances.
	SignalSet	3	The signals that can be received by this process instance. This set of signals is commonly known as the valid input signal set of the process instance. Each signal is described by its name followed by an SDT reference to the definition of the signal.
SAVE	InstRef	1	An SDT reference to the definition of this save.
	InType	1	The name of the state-containing entity in which this save is defined.
SERVICE	InstRef	1	An SDT reference to the definition of this service.

Entity class	Attribute	Category	Description
SERVICE_INST	InstRef	1	An SDT reference to the definition of this service instance.
	TypeRef	2	SDT references to service type definitions. The first reference is to the service type from which this service instance is instantiated followed by references to its supertypes all the way to the service type on top of the inheritance hierarchy.
SIGNALROUTE	InstRef	1	An SDT reference to the definition of this signalroute.
	SignalSet	3	The signals that are conveyed on this signalroute in the direction from the From-entity to the To-entity. Each signal is described by its name followed by an SDT reference to the definition of the signal.
	SignalSetRev	3	As SignalSet but for the signals that are conveyed in the opposite direction.
	From	1	The name of the process, process instance, service or service instance from which this signalroute starts. If the signalroute starts from the environment surrounding the block, block instance, process or process instance where this signalroute is located, this attribute is “env”.
	FromVia	1	This attribute is specified when the From-entity is a process instance or a service instance, and is then the name of the gate of that process instance or service instance to which this signalroute is connected.

Entity class	Attribute	Category	Description
	To	1	The name of the process, process instance, service or service instance to which this signalroute leads. If the signalroute leads to the environment surrounding the block, block instance, process or process instance where this signalroute is located, this attribute is “env”.
	ToVia	1	This attribute is specified when the To-entity is a process instance or a service instance, and is then the name of the gate of that process instance or service instance to which this signalroute is connected.
START[b]	InstRef	1	See the description for CONTINUOUS_SIGNAL.
	InType	1	See the description for CONTINUOUS_SIGNAL.
	Outputs	3	See the description for CONTINUOUS_SIGNAL.
	Calls	3	See the description for CONTINUOUS_SIGNAL.
	NextStates	3	See the description for CONTINUOUS_SIGNAL.
STATE	InstRef	2	SDT references to the state symbols in the state-containing entity that contain the name of this state. If the state-containing entity is a process type, service type or procedure, occurrences in the supertypes are also included.
SUBSTRUCTURE	ConnectionOut	2	The names of the channels that are on the outside of this substructure.

Entity class	Attribute	Cate-gory	Description
	ConnectionIn	2	The names of the channels or signal-routes that are on the inside of this sub-structure.
SYSTEM	InstRef	1	An SDT reference to the definition of this system.
SYSTEM_INST	InstRef	1	An SDT reference to the definition of this system instance.
	TypeRef	2	SDT references to system type definitions. The first reference is to the system type from which this system instance is instantiated followed by references to its supertypes all the way to the system type on top of the inheritance hierarchy.

a. The name of a CONTINUOUS_SIGNAL entity is set to the priority of the continuous signal. If no priority has been specified, maximum priority is assumed and the name is then set to “MAX_PRIO”.

b. The name of a START entity is always set to “Start”.

Information about the entities is generated in pre-order (prefix walk in the instance tree). This means that an entity is always followed by the entities defined within itself.

A formal and complete description of the general format of an instance information file would be rather complex, and is beyond the scope of this text. Instead, see Example 350:

Example 350: An instance information file

Consider the SDL system below:

```
System ExSystem;
  Signal Sig(Integer), Inp;
  channel ExChannel
    from ExBlock to env with Sig;
    from env to ExBlock with Inp;
  endchannel ExChannel;
  block ExBlock referenced;
endsystem ExSystem;

Block ExBlock;
  signalroute ExSignalRoute
    from ExProcess to env with Sig;
    from env to ExProcess with Inp;
  process ExProcess referenced;
  connect ExChannel and ExSignalRoute;
endblock ExBlock;

Process ExProcess;
  DCL Counter Integer;
  procedure ExProcedure referenced;
  start ;
    task Counter := 0;
    grst4:
  nextstate Wait;
  state Wait;
    input Inp;
    task {
      Counter := call ExProcedure(Counter);
    };
    output Sig(Counter);
    join grst4;
  endstate;
endprocess ExProcess;

Procedure ExProcedure;FPAR a Integer;RETURNS ret
Integer;
  start ;
  nextstate Idle;
  state Idle;
    input *;
    task {
      ret := a+1;
    };
    return ;
  endstate;
endprocedure ExProcedure;
```

The instance information file for this SDL system will look like this:

```
SDTINST V1.0
1  SYSTEM ExSystem
   InstRef = #SDTREF(TEXT,ExSystem.pr,1)
   .
2  CHANNEL ExChannel
   InstRef = #SDTREF(TEXT,ExSystem.pr,3)
   SignalSet = (
     Sig #SDTREF(TEXT,ExSystem.pr,2)
   )
   SignalSetRev = (
     Inp #SDTREF(TEXT,ExSystem.pr,2)
   )
   From = ExBlock
   To = env
   .
2  BLOCK ExBlock
   InstRef = #SDTREF(TEXT,ExSystem.pr,10)
   ConnectionOut = (
     ExChannel
   )
   ConnectionIn = (
     ExSignalRoute
   )
   .
3  SIGNALROUTE ExSignalRoute
   InstRef = #SDTREF(TEXT,ExSystem.pr,11)
   SignalSet = (
     Sig #SDTREF(TEXT,ExSystem.pr,2)
   )
   SignalSetRev = (
     Inp #SDTREF(TEXT,ExSystem.pr,2)
   )
   From = ExProcess
   To = env
   .
3  PROCESS ExProcess
   InstRef = #SDTREF(TEXT,ExSystem.pr,18)
   IniNoOfInst = 1
   SignalSet = (
     Inp #SDTREF(TEXT,ExSystem.pr,2)
   )
   .
4  START Start
   InstRef = #SDTREF(TEXT,ExSystem.pr,21)
   InType = ExProcess
   Nextstates = (
     Wait #SDTREF(TEXT,ExSystem.pr,24)
   )
   .
4  STATE Wait
   InstRef = (
     #SDTREF(TEXT,ExSystem.pr,25)
   )
   .
```

```
5  INPUT Inp
   InstRef = #SDTREF(TEXT,ExSystem.pr,26)
   InType = ExProcess
   Outputs = (
     Sig #SDTREF(TEXT,ExSystem.pr,30)
   )
   Calls = (
     ExProcedure #SDTREF(TEXT,ExSystem.pr,28)
   )
   Nextstates = (
     Wait #SDTREF(TEXT,ExSystem.pr,24)
   )
   .
4  CALLED_PROCEDURE ExProcedure
   InstRef = #SDTREF(TEXT,ExSystem.pr,20)
   .
5  START Start
   InstRef = #SDTREF(TEXT,ExSystem.pr,36)
   InType = ExProcedure
   Nextstates = (
     Idle #SDTREF(TEXT,ExSystem.pr,37)
   )
   .
5  STATE Idle
   InstRef = (
     #SDTREF(TEXT,ExSystem.pr,38)
   )
   .
6  INPUT Inp
   InstRef = #SDTREF(TEXT,ExSystem.pr,39)
   InType = ExProcedure
   Nextstates = (
     return #SDTREF(TEXT,ExSystem.pr,43)
   )
   .
```

Error Handling

Errors may be detected both during interaction with the user (command interpretation) and during analysis. Errors are displayed on the screen and optionally appended to an error file supplied by the user. When using the graphical UI, messages are directed to the Organizer log window.

Command Interpretation Errors

Errors that occur during command interpretation are errors due to:

- Illegal command parameters. See "Syntax of Analyzer Commands" on page 2475.
- Problems with accessing files on your computer system.

Diagnostics Issued During Analysis

Errors may be found during the various steps of the analysis. The different types of diagnostics and the format of the messages are described below. The messages are listed in "Error and Warning Messages" on page 2531 in chapter 54, *The SDL Analyzer*.

Diagnostic Types

Errors can be of the following types: warning, error, internal error, and information.

Warning

There are different types of warnings:

- Warnings that truncation has been performed, because the implementation limits have been exceeded.
- Warnings due to non implemented portions of the Analyzer.
- Warnings due to a badly designed SDL construction that is not essential for the interpretation.
- Warnings due to analysis actions that were not performed because the Analyzer could not recover from errors in previous analysis steps.

Error

One type of error is detected; violation of an SDL rule.

Internal Error

Internal errors due to malfunction of the Analyzer are detected. An internal error often has its roots in an SDL error from which the Analyzer did not recover properly.

Information

Often used to provide additional information to the other types of messages.

Diagnostic Format

Messages generally consist of the following parts:

- A reference to the source code that caused the diagnostic to be reported. See "Syntax" on page 917 in chapter 18, *SDT References*.
- The type of the diagnostic (ERROR / WARNING / INFO).
- A number identifying the diagnostic.
- A message describing the diagnostic.

Example of a Syntax Error

The message describing the error includes a text line where the illegal construct is indicated with a "?". If the trace back is common to many lexical or syntactic errors, the trace back will only occur after the last of these error messages.

The error also contains a reference to the source file.

Example 351: Syntax Error Produced by Analyzer ––––––––––––––––

```
#SDTREF(SDL,/usr/tom/game.spr(1),137(30,40),1,11)
ERROR 312 Syntax error in rule VARIABLE, symbol =
found but one of the following
expected:
!   (   :=
task Count=0 ;
          ?
```

––

The interpretation of the error message is that the Analyzer found the symbol "=" when it expected one of the symbols "!" (the field selector), "(" or ":=".

In the current example, the assignment statement can be found in position 11 the first line in the object with the ID 137 at position (30, 40) on page 3, on page 1 in the diagram stored on file `/usr/tom/game.spr`.

Example of a Semantic Error

The message describing the error includes, if possible, a line containing the error, with a "?" immediately preceding the SDL item that caused the error.

The error also contains a reference to the source file.

Example 352: Semantic Error Produced by Analyzer ––––––––––––––

```
#SDTREF(SDL,/usr/tom/game.spr(1),296(55,40),1)
ERROR 88 Undefined procedure
call ? ErrorHandler
```

––

The interpretation of this error message is that the procedure ErrorHandler is referred to but not is defined, and the call can be found in page 1 of the diagram stored on the on file `/usr/tom/game.spr`, in line 1 of the object with ID 296 and at position (55, 40).

Analyzer Files

When a diagram is being analyzed, a number of files are generated. The file names consist of the diagram name appended with a file name extension. The following file name extensions are used:

- `.pr`
  The SDL/PR, (also known as *phrasal representation*), of the SDL diagram. This file is generated by the conversion to PR pass. The purpose of this file is an intermediate format for the Analyzer. The format used in this file makes it not suitable to be read by the human.

- `.prm`
  After macros are expanded, the `.pr` file is used as input and expanded into a `.prm` file, which includes the SDL macros in an expanded form. The appearance of this file makes it not suitable to be read by the human.

- `.sdl`
  The pretty-printed SDL/PR file which is generated by the pretty-printer. This file uses a layout that is easy to read by the human.

- `.err`
  The error file containing the errors and warnings which were detected during the analysis.

- `.xrf`
  The files containing the listings of SDL definitions and cross-references.

- `.tsp`
  The file contains time stamp information from the last analysis.

- `.ins`
  The file containing instance information about the analyzed SDL system. Read more about the contents of this file in "SDL Instance Information" on page 2512.

- `predef.sdl`
  This file contains predefined SDL entities that the Analyzer needs access to in order to function properly. The file is found using the procedure in "Search Order for Included PR Files" on page 2506 with the directory `$sdtdir`.

Error and Warning Messages

This section contains a list of the Analyzer error and warning messages. Each message has a short explanation and, where applicable, a reference to the appropriate section of the recommendation Z.100, or to the formal definition. Z.100 is appended by the Formal Definition of SDL (Annex F.1-F.3) where a formal mathematical definition of the language is given.

Some messages contain a '%' followed by a number, which is used to indicate where information, specific to the error situation, will be included.

Some messages include a reference to the object that is the source of the diagnostic. These messages adhere to the format adopted in the SDL Suite. See chapter 18, *SDT References* for a reference to this format and for examples.

`INFO 0 Internal error: message %1 not found`

This message indicates an error in the implementation of the Analyzer. Please send a report to IBM Rational Customer Support, especially if the error can be reproduced as the only error message of an analysis. Contact information for IBM Rational Customer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*.

`ERROR 1 Internal error`

This message indicates an error in the implementation of the Analyzer. Please send a report to IBM Rational Customer Support, especially if the error can be reproduced as the only error message of an analysis. Contact information for IBM Rational Customer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*.

`ERROR 2 Cannot open file: %1`

This error message indicates that an error occurred when the Analyzer attempted to open a file. Modify, if necessary, the file protection and try to run the Analyzer again. If the error persists, contact IBM Rational Customer Support. Contact information for IBM Rational Customer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*.

ERROR 3 Cannot close file: %1

This error message indicates that an error occurred when the Analyzer attempted to close a file. Modify, if necessary, the file protection and try to run the Analyzer again. If the error persists, contact IBM Rational Customer Support. Contact information for IBM Rational Customer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*.

ERROR 4 Cannot delete file: %1

This error message indicates that an error occurred when the Analyzer attempted to delete a file. Modify, if necessary, the file protection and try to run the Analyzer again. If the error persists, contact IBM Rational Customer Support. Contact information for IBM Rational Customer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*.

ERROR 5 Cannot rename file: %1 to %2

This error message indicates that an error occurred when the Analyzer attempted to rename a file. Modify, if necessary, the file protection and try to run the Analyzer again. If the error persists, contact IBM Rational Customer Support. Contact information for IBM Rational Customer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*.

ERROR 6 Recursive definition of signal list %1

A signal list must not be based on itself, directly or indirectly.
(Z.100: 2.5.5)

ERROR 7 More than one %1 is visible in Packages

Several definitions with same name were found in a package. A qualifier might resolve the problem.
(Z.100: 2.4.1.2)

ERROR 8 Ending identifier (%2) must be equal to defining identifier (%1) in a referenced definition

When writing a definition, the name must be the same on the last row as on the first row.
(Z.100: 2.2.2)

ERROR 9 Referenced definitions are not unique (%1)

There are more than one definition which matches the reference.

You get this error if:

- You use the same diagram name for two different diagram references in one diagram.

To remove the error:

- Rename one of the referenced diagrams.

(Z.100: 2.4.1.3)

ERROR 10 Number of instances must be lexically equal in referenced process definition and process reference

(Z.100: 2.4.4)

ERROR 11 Different virtuality attributes in reference and referenced heading

(Z.100: 6.3.2)

ERROR 12 A referenced definition may not reference itself

It is not allowed to have a reference to itself in the reference definition.
(Z.100: 2.4.1.3)

ERROR 13 No referenced definition matches reference

A reference is found but not the corresponding definition.

You get this error if:

- You define a diagram as REFERENCED, but do not give a non-referenced definition of the diagram later.

To remove the error, either:

- Add a non-referenced diagram definition.
- Remove the diagram reference.

(Z.100: 2.4.1.3)

ERROR 14 Virtual start not allowed in operator definition

It is not allowed to have the start as virtual in an operator definition.

You get this error if:

- You use the keyword VIRTUAL for the start transition in an operator.

To remove the error:

- Remove the VIRTUAL keyword from the start transition.

(Z.100: 5.3.2)

INFO 15 Required item %1 is not connected to a file

The Organizer has ordered to analyze something depending on a diagram that is not connected to a file.

ERROR 16 Input %1 and output %2 should be different, aborting!

An Analyzer pass requires the input and output files to be different. Specify other input and/or output files.

ERROR 17 Definition name may not be qualified when not a referenced definition

A qualifier is only allowed on a definition if it is a referenced definition.

You get this error if:

- You use a qualifier when defining a diagram in place, i.e. without using the REFERENCED keyword.

To remove the error, either:

- Remove the qualifier.
- Use the keyword REFERENCED and move the definition of the child diagram out of the parent diagram.

(Z.100: 2.4.1.3)

ERROR 18 Referenced definition (%1) is not referenced in the Package definition

A reference to the definition (%1) is missing in the package definition. (Z.100: 2.4.1.3)

ERROR 19 Referenced definition (%1) is not referenced in the system definition

A reference to the definition (%1) is missing in the system definition.

You get this error if:

- You define a diagram outside the SDL system but do not reference it from the SDL system.

To remove the error, either:

- Introduce a reference construct in the SDL system to the diagram.
- Remove the unwanted diagram.

(Z.100: 2.4.1.3)

ERROR 20 Invalid separate analysis unit

It is not allowed to make a separate analysis of the unit selected. Start the analysis on the whole specification instead.

ERROR 21 Referenced substructure definition without name

It is not allowed to have a substructure without a name when referencing it or converting it to SDL/GR.
(Z.100: 3.2.2, 3.2.3)

ERROR 22 System definition may not occur as referenced definition to a system

Only system types may be referenced, not systems.
(Z.100: 2.4.1.3)

ERROR 23 Illegally placed process context parameter

A process context parameter is used where it is not allowed. Examples of bad use is in a system type or in a block type.
(Z.100: 6.2, 6.1.1.1, 6.1.1.2)

ERROR 24 Illegally placed procedure context parameter

A procedure context parameter is used where it is not allowed.
(Z.100: 6.2)

ERROR 25 Illegally placed signal context parameter

A signal context parameter is used where it is not allowed.
(Z.100: 6.2)

ERROR 26 Illegally placed timer context parameter

A timer context parameter is used where it is not allowed. For example in a system type, block type or service type.
(Z.100: 6.2, 6.1.1.1, 6.1.1.2, 6.1.1.4)

ERROR 27 Illegally placed variable context parameter

A variable context parameter is used where it is not allowed. For example in a system type or block type.
(Z.100: 6.2, 6.1.1.1, 6.1.1.2)

ERROR 28 Illegally placed synonym context parameter

A synonym context parameter is used where it is not allowed. For example in a signal.
(Z.100: 6.2, 2.5.4)

ERROR 29 Illegally placed remote procedure context parameter

A remote procedure context parameter is used where it is not allowed.
(Z.100: 6.2)

ERROR 30 Illegally placed remote variable context parameter

A remote variable context parameter is used where it is not allowed.
(Z.100: 6.2)

ERROR 31 Illegally placed sort context parameter

A sort context parameter is used where it is not allowed.
(Z.100: 6.2)

ERROR 32 Illegal redeclaration of name

All definitions in the same scope unit belonging to the same entity class must have different names, with some exceptions.

You get this error if:

- You use the same name as another item when declaring a new item.

To remove the error, either:

- Change one of the names in all appropriate places.
- Remove one of the declarations if the intention was to use the same variable.

(Z.100: 2.2.2)

ERROR 33 Illegally placed gate definition

A gate is defined where it is not allowed.
(Z.100: 6.1.4)

ERROR 34 %1 is not a time stamp file, will not save time stamps

On successful analysis the Analyzer saves info about this. Examine the file and rename or delete it to allow the Analyzer to save time stamps and possibly avoid unnecessary reanalysis in the future.

ERROR 35 Invalid regular interval

The character strings in a nameclass must be of length one and the first one smaller than the second.
(Z.100: 5.3.1.14)

ERROR 36 Integer literal expected

The name in the priority clause of a continuous signal must be an integer literal. This is also true for the name following a regular element in a name class literal.

You get this error if:

- You type something that is not an integer in a place where an integer is expected.

To remove the error:

- Replace the non-integer with an integer.

(Z.100: 4.11, 5.3.1.14)

ERROR 37 Invalid generator parameter %1

The actual generator parameter must match the type of the formal one.
(Z.100: 5.3.1.12.2)

ERROR 38 Illegal number of arguments

The number of actual parameters must be equal to the number of formal parameters when calling a procedure or creating a process. Similar rules apply for output, input and save of signals and timers.

ERROR 39 More than one default assignment; last from generator %1

A sort must not contain more than one default assignment. When instantiating many generators in a sort, only one of them may contain a default assignment.
(Z.100: 5.4.3.3)

ERROR 40 Recursive instantiation of generator %1

A generator definition must not instantiate itself, directly or indirectly.
(Z.100: 5.3.1.12.1)

ERROR 41 Undefined generator

Attempt to instantiate an undefined generator.
(Z.100: 5.3.1.12.2)

ERROR 42 No virtual definition %1 in super of enclosing definition

It is only allowed to redefine or finalize a definition if it is defined as virtual in the inherited type.

You get this error if:

- You are trying to redefine an entity that is not declared as VIRTUAL in the super type.

To remove the error, either:

- Make sure you are trying to redefine the correct entity.
- Make sure that there is a virtual entity to redefine in the super type.

(Z.100: 6.3.2)

ERROR 43 Ordering must not be defined more than once

All operator signatures in a sort must be different (ordering is shorthand for the relational operators).
(Z.100: 5.3.1.8)

ERROR 44 Name of formal parameter of class type must not equal name of generator %1

A generator must not have a formal parameter of class type with the same name as the generator definition.
(Z.100: 5.3.1.12.1)

ERROR 45 Formal generator parameters of classes literal and constant mixed

Literal and constant parameters must not have the same name.
(Z.100: 5.3.1.12.1)

ERROR 46 Only task, decision and transition option allowed in operator definition

You get this error if:

- You try to include signal sending in an operator.

To remove the error:

- Remove the signal sending.

(Z.100: 5.3.2)

ERROR 47 Only Join and Return allowed in operator definition

(Z.100: 5.3.2)

WARNING 48 Ending name (%1) in an asterisk state may lead to an error

The optional state name ending a state may be defined only if the state list is a single state name, in which case the ending state name should be that state name. The state list for an asterisk state is probably more than a single state name.
(Z.100: 2.6.3)

ERROR 49 A state with more than one state name in the state list cannot contain an ending name (%1)

The optional state name ending a state may be defined only if the state list is a single state name, in which case the ending state name should be that state name.
(Z.100: 2.6.3)

ERROR 50 Ending name (%2) must be equal to defining name (%1)

You get this error if:

- You use different names for a definition at the beginning and at the end of the same definition.

To remove the error, either:

- Change the name either at the beginning or at the end of the definition to get the same name in both places.
- Remove the optional name at the end.

(Z.100: 2.2.2, 2.6.3)

ERROR 51 Illegally placed virtuality

Virtual is used in a place where it is not allowed.

You get this error if:

- You use the keyword VIRTUAL for a transition that is not inside a type-based diagram.

To remove the error, either:

- Remove the VIRTUAL keyword from the transition.
- Make the diagram a type-based diagram.

ERROR 52 Illegally placed context parameter

The context parameter is placed where it is not allowed.
(Z.100: 6.2)

ERROR 53 Illegally placed virtuality constraint

The virtuality constraint is placed where it is not allowed.

You get this error if:

- You use the ATLEAST keyword for a type-based diagram that is not placed inside a type-based diagram.

To remove the error, either:

- Remove the ATLEAST construct.
- Move the type-based diagram with the ATLEAST construct to a type-based diagram.
- Make the diagram containing the type-based diagram with the ATLEAST construct a type-based diagram too.

ERROR 54 Illegally placed specialization

The specialization is placed where it is not allowed.

ERROR 55 Illegally placed instantiation

The instantiation is placed where it is not allowed.

ERROR 56 Packages can only contain type definitions

Systems, blocks, processes and services are not allowed, but their corresponding type are.
(Z.100: 2.4.1.2)

ERROR 57 Undefined package %1 in use clause

An used package is not defined. This error may also occur if you have two mutually dependent packages, which is not allowed. If you use ASN.1 modules or C headers, it may also occur if there are no dependency links between these and the SDL system.

You get this error if:

- You try to use a package that has not yet been declared.

To remove the error, either:

- Add a package definition named %1.
- Remove the use of the package.

ERROR 58 Illegally placed exported attribute

The exported attribute is placed where it is not allowed.

ERROR 59 Omitted actual context parameter

(Z.100: 6.2)

ERROR 60 Parameterized type %1 cannot be used as actual context parameter

A type with context parameters is not allowed to be used as an actual context parameter.
(Z.100: 6.2)

ERROR 61 Undefined actual context parameter

The actual context parameter is undefined or is not visible.
(Z.100: 6.2)

ERROR 62 Illegal number of actual context parameters (>)
(Z.100: 6.2)

ERROR 63 Illegal number of actual context parameters (<)
(Z.100: 6.2)

ERROR 64 Illegal number of actual context parameters (0)
(Z.100: 6.2)

ERROR 65 Formal context parameter %1 cannot be used as super type
It is not allowed to specialize a formal context parameter.
(Z.100: 6.2)

ERROR 66 Several matches
Resolution by context found several possible interpretations, use qualifiers to reduce the number of possibilities.
(Z.100: 2.2.2)

ERROR 67 No matching synonym or literal
You get this error if:

- You are using a synonym or literal of the wrong type. For instance, a constant expression of type integer may be required, but you supply a value of type real.

To remove the error, either:

- Replace the synonym or literal with a constant expression of the correct type.
- Define a synonym or literal of the correct type.

(Z.100: 2.2.2)

WARNING 68 Optional parameter of sort %1 omitted

A missing parameter may lead to uninitialized values, make sure it is not left out by mistake.

You get this warning if:

- You omit an IN parameter in a procedure call.

To remove the warning, either:

- Supply a constant value or a variable to the parameter in the procedure call.
- Remove the parameter from the procedure and from all calls to the procedure.

WARNING 69 Implicit remote variable, consider making it explicit

ERROR 70 Invalid import expression

The import identifier in an import expression is erroneous.

You get this error if:

- You use an integer where a boolean is expected.

To remove the error, either:

- Use a boolean value instead.
- Make the expected value an integer.

(Z.100: 4.13)

ERROR 71 Signal %1 is shadowed in connection

You get this error if:

- You have multiple declarations of a signal (same name) and two channels or signal routes that are connected uses different versions of a carried signal.

To remove the error, either:

- remove the inner declaration of the signal
- Rename one of the signals and determine which one the channels or signal routes in the connection should use.

ERROR 72 Invalid variable or attribute of variable

Reference to a nonexisting variable or a variable with illegal type or attribute from a view definition, a view expression or an export action.

You get this error either if:

- You EXPORT a variable that is not visible from the current scope.
- You EXPORT a variable that is not defined as EXPORTED.

To remove the error, either:

- Find the correct name of the variable you want to export.
- Make sure that the variable you want to export is defined as EXPORTED.

ERROR 73 Illegal use of parameterized signal

This message is output when context parameters are used in an SDL structure. Context parameters are not supported.

ERROR 74 Illegal use of parameterized sort

This message is output when context parameters are used in an SDL structure. Context parameters are not supported.

ERROR 75 Undefined timer

Use of an undefined timer in set, reset or active.

You get this error if:

- You are referring to a timer that has not been declared.

To remove the error, either:

- Declare the timer.
- Refer to an already existing timer instead.

(Z.100: 2.8)

ERROR 76 No range condition equals true in transition option

Exactly one of the branches out from a transition option must be true.

You get this error if:

- There is no alternative in a transition option that evaluates to true.

To remove the error, either:

- Change the condition for taking one of the alternatives to make sure that one alternative is always taken.
- Add a new alternative with a condition that evaluates to true.

(Z.100: 4.3.4)

ERROR 77 Undefined sort

A sort which is not defined is referenced.

You get this error if:

- You use a sort that has not been defined.

To remove the error, either:

- Use another sort that is already defined.
- Define the sort.

(Z.100: 2.2.2)

ERROR 78 Error not allowed where constant required

You get this error if:

- You use SDL keyword error in an expression required to be constant.

To remove the error:

- Replace the SDL keyword error with something else.

(Z.100: 5.3.9.1, 5.4.2.1)

ERROR 79 Variable expected

A variable was expected at this place.

You get this error either if:

- You use a constant value or an operator on the left side of the assignment.
- You give a constant value or the value of an operator instead of a variable as an actual parameter in a procedure call, when the corresponding formal parameter is an IN/OUT parameter.

To remove the error in the above cases, either:

- Use a variable on the left side of the assignment instead.
- Use a variable as an actual parameter in the procedure call instead.

You also get this error either if:

- You call a procedure using a constant as an actual in/out parameter.
- You call a procedure without specifying a parameter that is an in/out parameter.

To remove the error in the above cases, either:

- Change the constant or the not specified parameter to a variable of the correct type instead.
- Change the procedure to accept an in parameter instead of an in/out parameter.

(Z.100: 2.4.6)

ERROR 80 In parameter expected

An in parameter was expected according to the definition.

You get this error if:

- You are using an IN/OUT parameter in an imported procedure signature that should match an IN parameter in a remote procedure signature.

To remove the error, either:

- Change the imported procedure signature to use an IN parameter instead.
- Change the remote procedure signature to use an IN/OUT parameter instead.

ERROR 81 Unexpected parameter

The parameter should not be there according to the definition.

You get this error either if:

- You supply a parameter when you create a process that is not defined in an FPAR statement in the process definition.
- You add an extra parameter or return value to an imported procedure signature that should match a remote procedure signature.

To remove the error, either:

- Remove the extra parameter or return value.
- Add the extra parameter or return value to the definition of the called or created entity.

ERROR 82 Start transition required in instantiated type %1

An instantiated type must contain a start transition.

You get this error if:

- You try to instantiate a process type or a service type that has no start transition.

To remove the error, either:

- Instantiate a process type or a service type that has a start transition instead.
- Add a start transition to the process type or the service type.

ERROR 83 Start transition required in called procedure

You get this error if:

- You call a procedure that does not have a start transition.

To remove the error:

- Add a start transition to the procedure definition.

(Z.100: 2.4.6)

ERROR 84 Procedure call not allowed where constant required

You get this error if:

- You call a procedure in a place where a constant is required, such as in a variable or synonym definition.

To remove the error, either:

- Assign the variable a constant value in the variable definition.
- Assign the variable the return value of a procedure call in the start transition instead.

(Z.100: 5.3.1.9.1)

ERROR 85 Parameterized procedure cannot be called

(Z.100: 6.1.2)

ERROR 86 PId Expression or Process Identifier expected

An PId expression or a process identifier was expected at this place.

You get this error if:

- You supply a second parameter to import, but it is not a pid expression or a process identifier.

To remove the error, either:

- Change the second parameter to a pid expression or a process identifier.
- Remove the second parameter if it is not needed.

ERROR 87 Pid expression in procedure call only allowed for remote procedures

It is only allowed to have PId expressions in a procedure call if the procedure is a remote procedure.

You get this error if:

- You are using the TO keyword in a call to a procedure that is not remote and imported.

To remove the error, either:

- Remove the TO construct from the procedure call, if it is a normal procedure residing in this process.
- Make the procedure a REMOTE procedure in the other process, and declare the procedure as IMPORTED in this process.

(Z.100: 2.7.3, 4.14)

ERROR 88 Undefined procedure

Attempt to call an undefined procedure.

You get this error if:

- You call a procedure that is not defined or not visible from the current scope.

To remove the error, either:

- Make an existing procedure with the correct name visible from the current scope.
- Create an appropriate procedure if it does not exist.

(Z.100: 2.2.2, 2.7.3)

WARNING 89 Trailing parameter of sort %1 omitted

A missing parameter may lead to uninitialized values, make sure it is not left out by mistake.

ERROR 90 Two range conditions equals true in transition option

The branches out from a transition option must be mutually exclusive.

You get this error if:

- You have two alternatives in a transition option that are both evaluated to be true.

To remove the error, either:

- Change the condition for taking one of the alternatives to make sure that only one alternative is taken.
- Remove one of the alternatives that evaluated to true.

(Z.100: 4.3.4)

INFO 91 Location of the global definition previously mentioned

ERROR 92 Undefined remote variable

Attempt to use an undefined remote variable.

You get this error if:

- You use a remote variable that has not been defined.

To remove the error, either:

- Use another remote variable that is already defined
- Define the remote variable.

(Z.100: 4.13)

ERROR 93 Label expected on first item in free action

(Z.100: 2.6.7)

ERROR 94 Formal parameters required in operator definition

The operator definition must have formal parameters.

You get this error if:

- You have defined an operator that is missing a FPAR statement.

To remove the error:

- Introduce an FPAR statement in the definition of the operator.

(Z.100: 5.3.2)

ERROR 95 Recursive definition of parent sort %1 in syntype

A syntype must not be based on itself, directly or indirectly.

You get this error if:

- You define a syntype X that equals a syntype Y that equals syntype X.

To remove the error:

- Make sure that syntypes not equals each other in a circular chain, and that one syntype equals a non-syn-type.

(Z.100: 5.3.1.9)

INFO 96 Setlocale failed

The Analyzer failed to set the character handling part of the locale. It will remain in the default C locale.

ERROR 97 Undefined signallist

ERROR 98 New literal %1 must only occur once in literal renaming

All literals renamed in an inheritance must be given unique names.

You get this error if:

- You are declaring a newtype that inherits from another type, and renames two or more old literals to the same new name.

To remove the error:

- Make sure to use unique names for all renamed literals.

(Z.100: 5.3.1.11)

ERROR 99 Old literal %1 must only occur once in literal renaming

A literal must only be renamed once when inheriting.

You get this error if:

- You are renaming literals in a newtype that inherits from a super newtype, and the old literal names occur more than once in the list of renamings.

To remove the error:

- Make sure that the same old literal name only occurs once on the right side of the assignment in the list of renamings.

(Z.100: 5.3.1.11)

ERROR 100 Illegal redeclaration of operator signature

It is not allowed to have several operators with the same signature. This error may also occur if you use the same name twice in a sort with implicit operators, for example “structure”.

ERROR 101 Unexpected remote variable

Remote variable may not be used in [priority] input.

You get this error if:

- You are using a remote variable in a way that is not allowed, for instance as if it was a normal signal in an input construct.

To remove the error, either:

- Use the remote variable in a proper way instead: to update the value of the remote variable, use an IMPORT construct instead of trying to receive the value as if it was a signal.
- Change the remote variable to another more proper type for the intended use.

ERROR 102 Expression(s) in number of instances evaluates to error

(Z.100: 2.4.4)

ERROR 103 More than one sub signal is visible

You get this error if:

- You have two or more REFINED signals with the same name that is visible from the current place.

To remove the error:

- Rename (or remove) one of the REFINED signals, to get unique names.

(Z.100: 3.3)

ERROR 104 Undefined signal, timer, signallist, remote procedure or remote variable

No visible definition with proper entity kind found.
(Z.100: 2.2.2)

ERROR 105 Remote procedure definition missing

A definition of the remote procedure is missing.

You get this error if:

- You are referring to a procedure with an IMPORTED construct, but the procedure does not exist or is not defined as REMOTE.

To remove the error:

- Make sure there is a REMOTE definition of the procedure you are referring to in the IMPORTED construct.

ERROR 106 Recursive definition of synonym

The value of a synonym must not be based on itself, directly or indirectly.

ERROR 107 Gate must be connected

You get this error if:

- You instantiate a type without connecting all it’s gates.

To remove the error:

- Connect the gate.

(Z.100:6.1.4)

ERROR 108 Type mismatch for variable or formal parameter

ERROR 109 Signal refinement is not supported

ERROR 110 Only one varargs allowed

ERROR 111 Unexpected return value

The procedure has to be value returning to use return values.

ERROR 112 Return value expected

A value returning procedure must specify a return value.

You get this error if:

- You are returning from a procedure without specifying a return value, when the procedure is expected to return a value because of a RETURNS construct in the procedure heading.

To remove the error, either:

- Supply a return value in the RETURN construct.
- Remove the RETURNS construct from the procedure heading and remove any returned values from all RETURN constructs in the same procedure.

ERROR 113 Undefined state %1

A nextstate contains a name of a state which is not defined.
(Z.100: 2.6.7.2.1)

ERROR 114 Undefined label %1

Attempt to join an undefined label (connector).
(Z.100: 2.6.8.2.2)

ERROR 115 Open range does not match

A range condition does not match the actual context.
(Z.100: 5.3.1.9.1)

ERROR 116 Inherited operator %1 is not visible

Operators containing an exclamation cannot be inherited (and renamed).

You get this error if:

- You declare a newtype that inherits an operator that cannot be found in the current scope.

To remove the error in the above cases, either:

- Check that you are using the correct name for the operator.
- Declare the operator.

You also get this error if:

- You have a newtype that inherits from a super newtype, and you try to inherit an operator from the super newtype that does not exist.

To remove the error in the above case, either:

- Make sure that the name of the operator you wants to inherit is correct.
- Remove the name of the non-existing operator from the list of inherited operators.

(Z.100: 5.3.1.11)

ERROR 117 Old operator name %1 must only occur once in operator renaming

An operator must only be renamed once when inheriting.

You get this error if:

- You use the same name for the old operator name and the new operator name.

To remove the error:

- Use a new name instead of reusing the old name.

(Z.100: 5.3.1.11)

ERROR 118 New operator name %1 must only occur once in operator renaming

All operators renamed in an inheritance must be given unique names.

You get this error if:

- You use the same new operator name for several old operator names.

To remove the error:

- Make sure to use unique names for every new operator name.

(Z.100: 5.3.1.11)

ERROR 119 Inheritance for %1 is circular

You get this error if:

- You have created a circular chain of diagram inheritance.

To remove the error:

- Break the chain of circular inheritance by removing or changing the super diagram for one of the diagrams in the chain.

(Z.100: 6.3.1)

ERROR 120 Atleast constraint for %1 is circular

(Z.100: 6.3.2)

ERROR 121 Recursive sort inheritance

A sort must not inherit from itself, directly or indirectly.

You get this error if:

- You have created a circular chain of newtype inheritance.

To remove the error:

- Break the chain of circular inheritance by removing or changing the super newtype for one of the newtypes in the chain.

(Z.100: 5.3.1.11)

ERROR 122 Unexpected connect statement

A connect statement was not supposed to be found at this place.

INFO 123 %1 is one of the possible matches

ERROR 124 Signal expected in output

An output must contain at least one signal.

You get this error if:

- You are trying to send something from a process that is not a signal, such as a timer.

To remove the error, either:

- Change the name in the output construct to match a visible signal instead.
- Supply the value or variable that you were trying to send as a parameter to a signal.

(Z.100: 2.7.4)

ERROR 125 Undefined channel or signal route in output via

Identifiers used in a via clause of an output must be visible channels or signal routes.

You get this error if:

- You use a name of a non-existing channel or signal route after a VIA keyword in a signal sending.

To remove the error, either:

- Find a correct channel or signal route name and use that instead.
- Add the channel or signal route that you references in the VIA construct.
- Remove the VIA construct, if the VIA construct is not needed and the signal sending path can be determined uniquely anyway.

(Z.100: 2.7.4)

ERROR 126 Set with no time expression is only allowed for timers with default duration

It is only allowed to omit the time expression in set if there is a default duration specified for the timer.

You get this error if:

- You try to set a timer that does not have a default duration without specifying a duration in the SET construct.

To remove the error, either:

- Add a default duration to the timer definition:
  `TIMER myTimer:=10.`
- Specify a time in the SET construct: `SET(Now+10, myTimer).`

(Z.100: 2.8)

ERROR 127 Set must contain a time expression and a timer id

When using set you must specify when the timer should expire and which timer to set.
(Z.100: 2.8)

ERROR 128 Undefined process

Attempt to create an undefined process.

You get this error if:

- You are trying to create a process that has not been defined.

To remove the error, either:

- Check that you are using the correct name for the process that you want to create.
- Add a process definition of the process you want to create.
- Remove the process creation construct.

(Z.100: 2.2.2, 2.7.2)

ERROR 129 Created process must belong to the partition block %1

It is only allowed to create a dynamic instance of a process type if the process type definition is in the same block.

You get this error if:

- You try to create a process instance in another block than the block that the creating process resides in.

To remove the error:

- Make sure that the creating process and the process to be created resides in the same block diagram.
- Make sure that the creating process uses the correct name for the process to be created.

(Z.100: 2.7.2, FD)

ERROR 130 Remote procedure input and signal list input must not have parameters

It is not allowed to have parameters in a remote procedure input or when inputting a signal list.

You get this error if:

- You try to receive parameters by using a pair of parenthesis characters when referring to a remote procedure or a signal list in an INPUT construct.

To remove the error, either:

- Replace the remote procedure or the signal list with ordinary signals
- Remove the parameters and the associated pair of parenthesis characters.

(Z.100: 4.14)

WARNING 131 %1 : Signal not used in input or output

The defined signal has neither been used in an input nor in an output.

ERROR 132 Invalid procedure call

Expressions in Enabling condition and continuous signal must not contain procedure calls.

You get this error if:

- You are calling a procedure in an inappropriate place, such as in the expression related to a continuous signal.

To remove the error:

- Remove or move the procedure call from the inappropriate place.

ERROR 133 State name %1 in asterisk state list must be contained in other state lists

The state names used within the parenthesis (the exceptions) of an asterisk state must be defined somewhere in the enclosing body or the body of a supertype.

You get this error if:

- You include a name of a non-existing state in the list of exceptions after an asterisk (i.e. for all states) state.

To remove the error, either:

- Check that you are using the correct name for the state that should be excepted.
- Remove the unwanted exception.
- Introduce the missing state in your state machine.

(Z.100: 4.4)

ERROR 134 No state left to expand the asterisk with

The asterisk state list in the asterisk state includes all states in the enclosing process/procedure/service body.

You get this error if:

- You are using an asterisk (i.e. for all states) state that is representing zero states.

To remove the error, either:

- Make sure that you are not doing exceptions to the asterisk state for all states in the state machine.
- Replace the asterisk state with a normal state not using an asterisk.

(Z.100: 4.4)

ERROR 135 At least one state list must be different from asterisk

A process/procedure/service body cannot only contain asterisk states.

You get this error if:

- You only have asterisk states in the state machine.

To remove the error:

- Make sure that you have at least one non-asterisk state in the state machine.

(Z.100: 4.4)

ERROR 136 Several virtual continuous signals in a state with same or no priority

It is not possible to tell which transition would be affected by a redefinition.

You get this error if:

- You have several virtual continuous signals in the same state with the same priority.

To remove the error, either:

- Remove one of the virtual continuous signals.
- Give one of the virtual continuous signals another priority.

(Z.100: 6.3.3)

ERROR 137 Virtual continuous signal is virtual in super type with same or no priority

It is not possible to tell which transition would be affected by a redefinition.

You get this error if:

- You have a virtual continuous signal in both the sub type and the super type and the priority is the same or no priority is given.

To remove the error, either:

- Make the continuous signal in the sub type REDEFINED or FINALIZED instead of VIRTUAL.
- Give different priority to the continuous signal in the sub type than the priority in the super type.

(Z.100: 6.3.3)

ERROR 138 Redefined continuous signal is not virtual in super type with same or no priority

Cannot find the transition to redefine.

You get this error if:

- You redefine a continuous signal transition in a sub type and the same continuous signal transition cannot be found in a super type with the same priority.

To remove the error, either:

- If you intend to redefine an existing continuous signal, make sure the continuous signal you are redefining is starting from the same state and have the same priority.
- If you do not want to redefine an existing continuous signal, remove the REDEFINED or FINALIZED keyword.

(Z.100: 6.3.3)

ERROR 139 Filter exited with error code

A filter program terminated with a non zero exit code.

ERROR 140 External procedure not allowed

An external procedure cannot be mentioned in a <type expression>, in a <formal context parameter> or in an <atleast constraint>.

You get this error if:

- You try to use an external procedure where only a normal procedure is allowed.

To remove the error, either:

- Use a normal procedure instead of the external procedure.
- Make the external procedure a normal procedure.

ERROR 141 Number of block instances not allowed

Number of block instances may only be specified in typebased block definitions.
(Z.100: 6.1.3.3)

ERROR 142 No diagram to put definition in

It is not possible to convert something to a text symbol in SDL/GR without a surrounding diagram.

ERROR 143 Start is already defined in super type

It is not allowed to have more than one start transition and a start transition is already defined in the inherited type.

You get this error if:

- You redefine a transition from a super type without using the keyword REDEFINED or FINALIZED.

To correct the error, either:

- Use the keyword REDEFINED or FINALIZED for the redefined transition, if you intend to redefine the transition; make sure that the transition you redefine is either marked with VIRTUAL or REDEFINED.
- Remove the definition from the sub type if you do not intend to redefine it.

(Z.100: 2.6.2, 6.3)

ERROR 144 Start is defined as finalized in super type

It is not allowed to define the start transition as redefined, since it is defined as finalized in the inherited type.

You get this error if:

- You are trying to REDEFINE or FINALIZE the start transition in a sub type when it is declared as FINALIZED in a super type.

To remove the error, either:

- Change the start transition to VIRTUAL or REDEFINED in the closest super type where it is mentioned.
- Remove the start transition from the sub type if you did not intend to redefine the start transition.

(Z.100: 2.6.2, 6.3.3)

ERROR 145 Start is not defined as virtual in super

It is not allowed to define the start transition as redefined, since it is not defined as virtual in the inherited type.

You get this error if:

- You try to redefine the start transition in a sub type when the start transition is not defined as VIRTUAL in a super type.

To remove the error:

- Make the start transition VIRTUAL in the super type.

(Z.100: 2.6.2, 6.3.3)

ERROR 146 Multiple exits from state %2 with signal, timer or remote procedure %1

The signal, timer or remote procedure is contained in two inputs or one input and one save.

You get this error if:

- There are several ways to leave a state for the same signal, timer or remote procedure.

To remove the error:

- Remove enough ways to leave the state to make all remaining ways unique.

(Z.100: 2.6.3, 2.6.5)

ERROR 147 Virtual input %1 is already defined as input in state %2 in supertype

It is not allowed to define the input %1 as virtual, since it is already defined without virtual in the inherited type.

You get this error if:

- You are declaring a transition to be VIRTUAL in a sub type, when the transition is already VIRTUAL in the super type.

To remove the error:

- Use the keyword REDEFINED instead of VIRTUAL in the sub type.
- Make sure the transition is VIRTUAL in a super type.

(Z.100: 6.3.3)

ERROR 148 Redefined input %1 is not defined as virtual input in state %2 in supertype

It is not allowed to define the input %1 as redefined, since it is not defined with virtual in the inherited type.

You get this error if:

- You have a redefined transition in a sub type that does not exist in the super type.
- You have a redefined transition in a sub type that is not declared as virtual in the super type.

To remove the error, either:

- Make sure there is a matching virtual transition in the super type.
- Remove the REDEFINED keyword from the transition in the sub type.

(Z.100: 6.3.3)

ERROR 149 Redefined input %1 is defined as finalized input in state %2 in supertype

It is not allowed to define the input %1 as redefined, since it is defined as finalized in the inherited type.

You get this error if:

- You are trying to REDEFINE a transition that is marked as FINALIZED in a super type.

To remove the error, either:

- Change the transition from FINALIZED to REDEFINED in the super type.
- Make sure that you are redefining the correct transition.

(Z.100: 6.3.3)

ERROR 150 Redefined input %1 is defined as input in state %2 in supertype

It is not allowed to define the input %1 as redefined, since it is defined without virtual in the inherited type.

You get this error if:

- You define a transition as REDEFINED in a sub type, when the same transition in the super type is not VIRTUAL.

To remove the error, either:

- Make the transition in the super type VIRTUAL.
- Make sure that you are redefining the correct transition.

(Z.100: 6.3.3)

ERROR 151 Finalized input %1 is not defined as virtual input in state %2 in supertype

It is only allowed to define an input as finalized if it is a virtual defined input in a inherited type.

You get this error either if:

- You use the keyword REDEFINED or FINALIZED for a transition that is not declared VIRTUAL in a super type.
- There is no super type.

To remove the error, either:

- Make the transition in the super type VIRTUAL.
- Change the name in the sub type to match the correct transition in the super type
- Make sure the sub type inherits from the correct super type.

ERROR 152 Finalized input %1 is defined as finalized input in state %2 in supertype

It is not allowed to define the input %1 as finalized, since it is already defined as finalized in the inherited type.

You get this error if:

- You try to finalize a transition in a sub type that is already FINALIZED in a super type.

To remove the error, either:

- Change the keyword FINALIZED to REDEFINED in the super type.
- Make sure that you are trying to finalize the correct transition in the sub type.

(Z.100: 6.3.3)

ERROR 153 Signal %1 is already defined as input in state %2 in supertype

It is not allowed to have the same signal in more than one input to the same state. In this case there is already a input of the signal %1 in the type you inherit from.

You get this error if:

- You try to define a transition that already exists in a super type, without using the keyword REDEFINED or FINALIZED, or when the transition in the super type is not virtual.

To remove the error:

- Add the keyword REDEFINED or FINALIZED to the transition in the sub type.
- Make sure that the transition in the super type is VIRTUAL.

ERROR 154 A state (%1) cannot contain both asterisk input and asterisk save

It is not allowed to both have a asterisk input and a asterisk save to a state.

You get this error if:

- You have a state in your state machine that has both an asterisk input transition and an asterisk save.

To remove the error, either:

- Remove the asterisk save.
- remove the asterisk input transition.

(Z.100: 4.6, 4.7)

ERROR 155 A state (%1) may only contain one asterisk input

It is only allowed to have one asterisk input to a state.

You get this error if:

- You are having more than one asterisk input transition for the same state.

To remove the error:

- Remove all but one asterisk input transition for the same state.

(Z.100: 4.6)

ERROR 156 A state (%1) may only contain one asterisk save

It is only allowed to have one asterisk save to a state.

You get this error if:

- You have a state in your state machine that has more than one asterisk saves.

To remove the error:

- Reduce the number of asterisk saves for the transition to one (or zero).

(Z.100: 4.7)

ERROR 157 Valid input signal set must be specified (in process/service %1) when no signal routes or channels are specified in the enclosing block/process (%2)

If there are no signal routes or channels specified to the process/service. “Signalset” must be used inside the process/service extended heading to specify the valid input signal set.

You get this error if:

- You have a process or service that is not connected with the outer world via signal routes or channels, and you have not specified an input SIGNALSET.

To remove the error, either:

- Add signal routes and channels to connect the process to the outer world.
- Add a SIGNALSET construct to the process and indicate the signals the process can receive.

(Z.100: 2.5.2)

ERROR 158 Value returning procedure expected

A value returning procedure was expected at this place.

ERROR 159 Type mismatch: sort %2 does not match %1

You get this error if:

- You are using an integer where a boolean is expected. For instance, you are using an integer as a type for a parameter in an imported procedure signature that should match a boolean type parameter in a remote procedure signature.

To remove the error, either:

- Use a boolean instead.
- Change the expected type to be an integer instead.

ERROR 160 In/out parameter expected

An in/out parameter was expected according to the definition.

You get this error if:

- You are using an IN parameter in an imported procedure signature that should match an IN/OUT parameter in a remote procedure signature.

To remove the error, either:

- Change the imported procedure signature to use an IN/OUT parameter instead.
- Change the remote procedure signature to use an IN parameter instead.

ERROR 161 Missing parameter

A parameter is missing according to the definition.

You get this error if:

- You omit a parameter or return value from an imported procedure signature that should match a remote procedure signature.

To remove the error, either:

- Add the missing parameter or return value to the imported procedure signature.
- Remove the not wanted parameter or return value from the remote procedure signature.

ERROR 162 Virtual definitions only allowed in types

It is only allowed to use virtual in a type definition.

You get this error if:

- You have a virtual definition of a diagram in a normal diagram.

To remove the error:

- Remove the keyword virtual from the definition of the diagram.
- Make the diagram with the virtual definition a type-based diagram.

ERROR 163 Virtual definition %1 does not conform to its virtuality constraint

(Z.100: 6.3.2)

ERROR 164 Redefinition of %1 does not conform to the virtuality constraint

(Z.100: 6.3.2)

ERROR 165 A gate with addition of signals can only occur in subtypes

Adding in gate definition can only be used if the type is inherited from another type.

You get this error if:

- You are using the ADDING construct for a gate in a diagram that does not inherit from another diagram.

To remove the error, either:

- Remove the ADDING construct from the gate definition.
- Make the type-based diagram with the error inherit from another type-based diagram with an appropriate gate that you can use.

(Z.100: 6.1.4)

ERROR 166 Definition of %1 exists already in super type

An already defined gate cannot be defined again. If you want to add signals to an already existing gate, the word adding must be used.

You get this error if:

- You declare a gate in a type that inherits from another type that already has a gate with the same name.

To remove the error in the above case:

- Rename one of the gates to get two gates in the sub type.
- Use the keyword ADDING in the gate in the sub type if you want to redefine the gate in the super type from the sub type.

You also get this error if:

- You use the RETURNS construct in both the super procedure type and the sub procedure type.

To remove the error in the above case:

- Remove the RETURNS construct from the sub type.

You also get this error either if:

- You try to redefine a finalized procedure in a sub type.
- You try to define a procedure, that has already been defined as virtual in a super type, without using REDEFINED.

To remove the error in the above cases, either:

- Make sure that you are redefining the correct procedure.
- Make sure that you use the REDEFINED keyword in the sub type, if you intend to redefine a procedure.
- Change the name of the procedure in the sub type, if you do not intend to redefine any existing procedure.
- Change the finalized procedure in the super type to be a redefined procedure.

`ERROR 167 A gate with addition of signals must already be defined in a super type`

Adding in gate definition cannot be used when defining a new gate.

You get this error if:

- You declare a gate endpoint constraint with the keyword ADDING, but the gate is not already declared in any super type.

To remove the error, either:

- Make sure that the name of the gate with the ADDING keyword matches the name of a gate in a super type.
- Remove the ADDING keyword to get a new gate that does not depend on any gate in a super type.

(Z.100: 6.1.4)

ERROR 168 Two gate constraints may not use the same direction in gate %1

It is not allowed to have the same direction (in or out) in two gate constraints.

You get this error if:

- You have two gates in the same type-based diagram going in the same direction.

To remove the error, either:

- Reverse one of the gates.
- Merge the two gates, i.e. remove one of them, if they are intended to go in the same direction.

(Z.100: 6.1.4)

ERROR 169 Undefined type %1 in gate endpoint constraint

The endpoint constraint in the gate contains a undefined or not visible type.
(Z.100: 6.1.4)

ERROR 170 The two endpoint constraints of gate %1 is not consistent

The endpoint constraints, to and from, in the gate are not consistent.

You get this error if:

- You have two gate endpoint constraints in a type-based diagram that are going in the same direction.
- You have two gate endpoint constraints in a type-based diagram that are not referencing the same outer entity.

To remove the error:

- Make sure that the two gate endpoint constraints are going in different directions.
- Make sure that if you reference an outer type-based diagram, the same type-based diagram is referenced in both gate endpoint constraints.

(Z.100: 6.1.4)

ERROR 171 Undefined block, channel or signal route in connection

Identifiers used in a connection must be visible in the scope and of the above mentioned types.
(Z.100: 2.5.3, 3.2.3)

ERROR 172 Undefined block, process or service in path

The identifiers following the text “from” and “to” in channel and signal route definitions must be visible blocks, processes or services.
(Z.100: 2.5.1, 2.5.2)

ERROR 173 Selected diagram not in an SDL system

The context must be available when analyzing part of a system.

ERROR 174 Undefined type %1 in instance specification

The type must be defined and visible in the scope to be instantiated.

You get this error if:

- You refer to a type-based diagram that does not exist.

To remove the error, either:

- Refer to an existing type-based diagram instead.
- Create a matching type-based diagram.

ERROR 175 Undefined component %1/%2 in use clause

(Z.100: 2.4.1.2)

ERROR 176 Parameterized type %1 cannot be used as constraint type

It is not allowed to use a type with context parameters as a constraint type.
(Z.100: 6.3.2)

ERROR 177 Undefined type %1 in constraint specification

The type %1 used in the constraint specification is undefined or not visible.

ERROR 178 Substructure name in subtype and supertype must be the same

(Z.100: 6.3.1)

ERROR 179 Undefined type %1 in inheritance specification

A type which is inherited is not defined or not visible.

You get this error if:

- You in a signal declaration uses the keyword INHERITS and mention another signal that has not yet been defined.

To remove the error:

- Make sure the signal to inherit from is defined before the signal that inherits.

ERROR 180 Undefined type %1 in virtuality constraint

The type %1 used in the virtuality constraint is undefined or not visible.

You get this error if:

- You refer to a type-based diagram that does not exist in a virtuality constraint.

To remove the error, either:

- Use the name of an existing and matching type-based diagram instead.
- Create a matching type-based diagram.
- Remove the virtuality constraint.

ERROR 181 Environment must only appear in one connection

When a channel is partitioned and has the environment as one of its endpoints, the channel substructure must contain one and only one connection with environment.

You get this error if:

- You have more than one connect statement for a signal route or channel going to or from the environment.

To remove the error:

- Remove all but one connect statement going to or from the environment for the signal route or channel in question.

(Z.100: 3.2.3)

ERROR 182 The path %1 must only appear in one connection

It is not allowed to have the same channel or signalroute name mentioned in more than one connection.

You get this error if:

- You mention a signal route or a channel in two or more CONNECT statements in the same diagram.

To remove the error:

- Merge all CONNECT statements mentioning the same signal route or channel into one CONNECT statement.

(Z.100: 2.5.3, 3.2.2)

ERROR 183 All subsignals to the channel %1 must be included in the signal lists of the subchannels

In the connection between a channel and subchannels, each signal conveyed by the channel must either be conveyed by the subchannels or all of its subsignals must be conveyed by the subchannels.
(Z.100: 2.5.3, 3.3)

ERROR 184 Signal %1 must be included in the signal list of the inner paths

In a connection point, all signals in the outer paths must be conveyed by at least one of the inner paths.

You get this error if:

- You connect two signal routes or channels that do not allow the same set of signals to be sent in each direction.

To remove the error:

- Add or remove signals from the signal lists in the connected signal routes or channels to get matching signal lists.

(Z.100: 2.5.3, 3.3)

ERROR 185 Signal %1 must be included in the signal list of the outer paths %2

In a connection point, all signals in the inner paths must be conveyed by at least one of the outer paths.

You get this error if:

- You connect a signal route or channel to a signal route or channel one level up in the diagram hierarchy that does not include all signals of the signal route or channel in this diagram.

To remove the error, either:

- Add the signals you want to send to the appropriate signal lists in the diagrams outside this diagram.
- Remove the not wanted and troublesome signal from the signal list of the channel or signal route that is connected to the world outside this diagram.

(Z.100: 2.5.3, 3.3)

ERROR 186 In the implicit connection with %2 signal %1 is not included in any of the outgoing implicit signal routes

Each signal in the channels, directed out from the block, must be mentioned in at least one output in a process. See also *ERROR 187*.
(Z.100: 2.5.2, 2.5.3)

ERROR 187 In the implicit connection with %2 is signal %1 not included in any of the incoming implicit signal routes

When a block contains no signal routes, the signal route definitions and the connections are derived from the valid input signal sets and the outputs (of the processes in the block) and the channels connected to the block. (The same is valid for a service decomposition with no service signal routes.) Each signal in the channels, directed into the block, must be included in at least one valid input signal set.
(Z.100: 2.5.2, 2.5.3)

ERROR 188 Actual context parameter %1 does not conform to the fpar constraint

The actual context parameter is not correct according to the fpar constraint given for the formal context parameter.
(Z.100: 6.2)

ERROR 189 Actual context parameter %1 does not conform to the sort signature constraint

The actual context parameter is not correct according to the sort signature constraint given for the formal context parameter.
(Z.100: 6.2)

ERROR 190 Actual context parameter %1 does not conform to the atleast constraint

The actual context parameter is not correct according to the at least constraint given for the formal context parameter.
(Z.100: 6.2)

ERROR 191 Actual context parameter %1 does not conform to the process type constraint

The actual context parameter is not correct according to the process type constraint given for the formal context parameter.
(Z.100: 6.2)

ERROR 192 Actual context parameter %1 does not conform to the sort list constraint

The actual context parameter is not correct according to the sort list constraint given for the formal context parameter.
(Z.100: 6.2)

ERROR 193 Actual context parameter %1 does not conform to the signal set constraint

The actual context parameter is not correct according to the signal set constraint given for the formal context parameter.
(Z.100: 6.2)

ERROR 194 Block definition must contain process, block or substructure definition

A block definition must at least contain one process or a substructure.
(Z.100: 2.4.3)

ERROR 195 Block definition must contain process definition(s) when containing signal route definition(s)

A block definition cannot contain signal routes when no processes are defined.
(Z.100: 2.4.3)

ERROR 196 Block definition must contain process or block definition(s) when containing connection(s)

A block definition cannot contain connections when no processes are defined.
(Z.100: 2.4.3)

WARNING 197 Expansion of select definition is not yet implemented

The SDL concept select definition is not supported by the SDL Analyzer.

> **Caution!**
>
> A consequence of this is that the parts of your SDL system which are contained in the select definition will **not** be analyzed or be input to the SDL to C compilers.

ERROR 198 Substructure definition must contain at least one block definition

At least one block must be defined inside a substructure.

You get this error if:

- You have a substructure diagram that does not contain any block diagrams.

To remove the error, either:

- Add at least one block diagram with one or several processes to the substructure diagram.
- Instantiate a block type diagram in the substructure diagram.

(Z.100: 3.2.2)

ERROR 199 Gate only allowed in paths with instance of a type or env in a type

Gates are only allowed when connecting channels or signal routes to types.

You get this error if:

- You are using gates without using type-based diagrams.

To remove the error in the above case, either:

- Remove the reference to the gate.
- Make the current diagram type-based, if you want to have gates to connect to in the environment.
- Make referenced diagrams instances of type-based diagrams, if you want to have gates to connect to in reference symbols.

You also get this error if:

- You use the VIA keyword when declaring a signal route or a channel, that is not going to or coming from an instantiation of a type-based diagram, and that is not going to or coming from the environment in a type-based diagram.

To remove the error in the above case:

- Remove the VIA construct.

(Z.100: 6.1.4)

ERROR 200 Undefined gate

Use of an undefined gate in a channel or signal route definition.

You get this error if:

- You are referring to a gate that does not exist.

To remove the error, either:

- Remove the gate reference.
- Add the gate you are referencing.

ERROR 201 Connected to gate (%1) in wrong scope

(Z.100: 6.1.4)

ERROR 202 Connected instance %1 does not conform to the endpoint constraint of gate %2

The type of the instance %1 connected to gate %2 must be equal to or be a subtype of the endpoint constraint of gate %2.

ERROR 203 ENV does not conform to the endpoint constraint of gate %1

A gate with an endpoint constraint cannot be connected to the environment.

You get this error if:

- You have a gate endpoint constraint in the type you try to instantiate, and you connect the gate to an instance that is not of the same type (or a descendant of that type) as the one mentioned in the gate endpoint constraint.

To remove the error, either:

- Connect the gate to an instance of the same type (or a descendant of that type) as the one mentioned in the gate endpoint constraint.
- Remove or change the gate endpoint constraint.

ERROR 204 Signal %1 in path is not an incoming signal of %3 gate %2

The gate %2 of instance %3 must be defined to convey the incoming signal %1. Add the signal to the gate definition.

ERROR 205 Signal %1 in path is not an outgoing signal of %3 gate %2

The gate %2 of instance %3 must be defined to convey the outgoing signal %1. Add the signal to the gate definition.

ERROR 206 Via gate expected in env path with block type or process type

A channel or a signal route must be defined using “via <gate>” when connected to the environment in a block type or system type.

You get this error if:

- You declare a channel or a signal route in a type-based diagram going to or coming from the environment without using a VIA keyword to specify the gate to use.

To remove the error:

- Add a VIA keyword and mention the gate to use in the declaration of the signal route or channel.

ERROR 207 Via gate expected in path with typebased instance

Defining a channel or a signal route using “via <gate>” is only possible when the connected instance is type based and thus contain gates.

You get this error if:

- You declare a signal route or a channel going to or coming from an instantiation of a type-based diagram, without specifying the gate to connect to with a VIA keyword.

To correct the error:

- Add a VIA keyword and the name of the gate in the instantiated type-based diagram that you want to connect to.

ERROR 208 Endpoints of the path %1 must be different

A channel/signal route cannot connect a block/process/service with itself.

You get this error if:

- You specify the same endpoint for both ends of a signal route or channel.

To remove the error:

- Connect one of the ends of the signal route or channel to another endpoint.

(Z.100: 2.5.1, 2.5.2)

ERROR 209 Endpoint %1 of channel %2 must be a block or process

A channel must be connected to at least one block!

(Z.100: 2.5.1)

ERROR 210 Endpoint %1 of the path %2 must be locally defined

The endpoints of a channel/signal route must be defined in the same scope as the channel/signal route is defined.

You get this error if:

- You try to connect a signal route or channel to a diagram that is not visible from the current diagram.

To remove the error:

- Connect the signal route or channel to the environment or to a diagram one level below the current diagram. If necessary, create more signal routes or channels in other diagrams to reach the diagram you want to communicate with.

(Z.100: 2.5.1, 2.5.2)

ERROR 211 The second path must denote reverse direction of the first path in %1

In a bidirectional channel or signal route, the second path must be in the reverse direction of the first path.

You get this error if:

- The second FROM-TO-VIA-WITH construct is not going in exactly the reverse direction as compared to the first one.

To remove the error, either:

- Make sure that the same connection item is used after both FROM in the first construct and after TO in the second construct.
- Make sure that the same connection item is used after both TO in the first construct and after FROM in the second construct.

(Z.100: 2.5.1, 2.5.2)

ERROR 212 Endpoint %1 of signal route %2 must be a process or a service

A signal route must connect at least one process or service.
(Z.100: 2.5.2)

ERROR 213 Return not allowed here

You get this error if:

- You use the return construct in a process diagram.

To remove the error, either:

- Replace the return with a stop.
- Make the process diagram a procedure diagram.

(Z.100: 2.6.8.2.4)

ERROR 214 Process definition must contain either processbody or service decomposition

Syntax requirement.
(Z.100: 2.4.4)

ERROR 215 Process cannot contain timer definition when decomposed into services

(Z.100: 2.4.5)

ERROR 216 Revealed and exported variables only allowed in process and service

(Z.100: 2.4.6)

ERROR 217 Complete semantic analysis requires a system

(Z.100: 2.4.6)

ERROR 218 Procedure cannot contain stop

Syntax requirement.
(Z.100: 2.6.8.2.3)

ERROR 219 Undefined view

ERROR 220 Step expression requires a loop variable

It is not allowed to omit the variable indication from a for statement when having a step expression.

You get this error if:

- You have a for(x,y,z)-statement where x does not define a loop variable, but z uses a loop variable.

To remove the error, either:

- Define a loop variable in the for statement.
- Change the step expression to match the loop variable.

ERROR 221 Break without name or continue are only allowed within a loop

Break terminates a for loop and continue does the next iteration of a for loop. They can only be used in the context of for loops.

You get this error either if:

- You use a BREAK construct without a name outside a loop.
- You use a CONTINUE construct outside a loop.

To remove the error, either:

- Add a name after the break keyword that matches a label that you want to jump to.
- Remove the break or continue construct.
- Put the break or continue construct within a for loop.

ERROR 222 Variable used before it is defined

A variable may only be used in statements following its definition.

You get this error if:

- You use a variable before the declaration of the variable.

To remove the error, either:

- Move either the variable use or declaration to make sure that the variable declaration comes before the variable use.
- Make sure you are using the correct variable.

ERROR 223 A jump statement must be contained in a labeled statement with the given name

A jump statement is a more restrictive form of a join statement. The intent is to create less convoluted code.

You get this error if:

- You have a break construct to a label that is not visible from the break.

To remove the error:

- If the label and the break are on the same level, begin a pair of brackets after the label that contains the break.

ERROR 224 No receiver found

ERROR 225 Two services cannot have the same signal (%1) in their valid input signal set

The complete valid input signal sets of the service definitions within a process definition must be disjointed.
(Z.100: 2.4.5)

ERROR 226 Recursive package %1 in use clause

A package must not use itself.

You get this error if:

- There is a self-referential chain of uses of packages. For instance package A uses package B, and package B uses package A.

To remove the error:

- Break the self-referential chain of uses of packages by removing a USE statement in an appropriate place in the chain.

ERROR 227 Ending type keyword must match starting

A process type must be terminated with endprocess type and not just endprocess, and so on.

ERROR 228 Not Charstring sort and no proper operator (Length, ...)

ERROR 229 Service signal routes cannot be specified when no signal routes are specified in the enclosing block

(Z.100: 2.4.5)

ERROR 230 Service decomposition must contain at least one service definition

(Z.100: 2.4.4)

ERROR 231 No proper Length operator found

ERROR 232 Recursive include of file %1

The file is included recursively.

ERROR 233 Qualifier in component command not in SDL system

All parts (components) of a program must belong to the same system.

ERROR 234 More than one system

Can only analyze one system at a time.

ERROR 235 Code generation requires a system

Packages are not sufficient.

ERROR 236 Valid input signal set cannot contain a timer (%1)

You get this error if:

- You include a timer in the specification of the set of signals that a process can receive.

To remove the error:

- Remove the timer from the signal set construct.

(Z.100: 2.4.4, 2.4.5)

ERROR 237 Two signals (%1 and %2) in the complete valid input signal set are on different refinement levels of the same signal

(Z.100: 3.3)

ERROR 238 Two signals (%1 and %2) among the output-signals of the process are on different refinement levels of the same signal

(Z.100: 3.3)

ERROR 239 Signal %1 in save cannot be received

The signal is not included in the complete valid input signal set of the process/service.
(FD)

ERROR 240 Signal %1 in input cannot be received

The signal is not included in the complete valid input signal set of the process/service.

You get this error if:

- You try to receive a signal in an input, but you have not declared that anyone can send that signal to this process.

To remove the error, either:

- Check that you are trying to receive the correct signal.
- Make sure that the signal can be sent to this process via signal routes and channels, or by including the signal in a SIGNALSET construct for this process.

(FD)

ERROR 241 Error creating directory %1

A message from the operating system is following this and hopefully clarifies the problem.

ERROR 242 Signal %1 in priority input cannot be received

The signal is not included in the complete valid input signal set of the service.
(FD)

ERROR 243 Actual in/out param sort %1 does not match %2

The sort of actual and formal in/out parameters of procedures must be identical on the syntype level.

You get this error if:

- You call a procedure using a parameter that is not of the correct type.

To remove the error, either:

- Call the procedure with a parameter of the correct type.
- Change the procedure to accept a parameter of the wanted type.

ERROR 244 Call procedure from expression or supply variable parameter

A value returning procedure should be called from an expression.

You get this error if:

- You call a value-returning procedure from outside of an expression.

To remove the error, either:

- Call the value-returning procedure from an expression in a task instead.
- Call another, similar procedure that do not return a value.
- Force the procedure to not return a value.

WARNING 245 Trailing parameter of sort %1 omitted

ERROR 246 Last param of procedure called from expr is not in/out

Actual in/out procedure parameters may not be omitted.

You get this error if:

- You are calling a procedure in an expression, and the called procedure does not have a RETURNS statement.

To remove the error:

- Add a RETURNS statement to the procedure definition, and make sure the procedure returns a value of the correct type.

ERROR 247 Actual in/out param cannot be omitted

The actual parameter can be omitted if the formal is in.

You get this error if:

- You call a procedure without less actual parameters than all formal parameters declared for the procedure.

To correct the error, either:

- Add the missing actual parameters to match all formal parameters declared for the procedure.
- Remove any not wanted formal parameters from the declaration of the procedure.

WARNING 248 License %1 will expire in %2 days

This is a notification that some license will expire in the near future.

ERROR 249 Aborted by exit request

This indicate that the analyzer received a stop message

ERROR 250 Component and Thread commands requires a previous Program command

Issue a program command to set the name of the executable before adding threads or components to it.

ERROR 251 Change directory: %1: %2

An error occurred when changing default (working) directory.

ERROR 252 Get work dir: %1: %2

The analyzer failed to obtain current working directory.

ERROR 253 Connection with %1 is missing

Either a path is connected to a block/process but is not contained in a connection or a block is connected to a channel but is not contained in a connection in the channel substructure.
(Z.100: 2.5.3, 3.2.2, 3.2.3)

ERROR 254 Connection with environment is missing

When a channel is partitioned and has the environment as one of its endpoints, the channel substructure must contain a connection with the environment.

You get this error if:

- You omit a connect statement saying that a channel or signal route is connected to the environment, when the declaration of the channel or signal route indicates that it is connected to the environment.

To correct the error:

- Add a connect statement saying that the channel or signal route is connected to the environment.

(Z.100: 3.2.3)

ERROR 255 Transition must end with a terminator

If the terminator of a transition is omitted, then the last action in the transition must contain a terminating decision.
(Z.100: 2.6.4, 2.6.8.1)

ERROR 256 Post Master

A problem occurred when communicating with some other tool using the PostMaster.

WARNING 257 Statement not reached

SDL-92 allows dead code in transition. The warning is issued to notify the user of the dead code.

ERROR 258 Initial transition ends (directly or indirectly) with dash nextstate

A nextstate in the initial transition (of a process/ procedure/ service) must be specified with a state name.
(Z.100: 4.9)

ERROR 259 Ending name (%1) cannot be qualified (except for remote definitions)

You get this error if:

- You are using a qualifier when specifying the name of the diagram at the end of the diagram definition.

To remove the error:

- Remove the qualifier and keep just the name of the diagram.

(Z.100: 2.2.2)

ERROR 260 The outer channeldef must have the first connectionpoint (environment) as one of the endpoints

A channel substructure can only contain a connection with the environment if the partitioned channel is connected to the environment.
(Z.100: 3.2.3)

ERROR 261 Answers are not mutually exclusive

Exactly one answer in a decision must match the question. For instance, a boolean decision with more than 2 answers or 2 answers that are both true or both false will produce this message.

You get this error if:

- It is possible to take two or more paths after the decision for at least one given set of values.

To remove the error:

- Make sure that it is always only possible to take one path after the decision symbol.

ERROR 262 The outer channeldef must have the first connectionpoint (block) %1 as one of the endpoints

The block identifier in a connection in a channel substructure must identify one of the endpoints of the partitioned channel.
(Z.100: 3.2.3)

ERROR 263 One of the endpoints of path %1 must be the scope unit %2

A path that occurs as the first connection point in a connection, must be connected to the block/process that contains the connection.
(Z.100: 2.5.3, 3.2.2)

ERROR 264 The second connectionpoint %1 must be defined in the scope unit %2

A path occurring in the second part (i.e. the list of subpaths) in a connection must be locally defined.

You get this error if:

- You have a connection statement connecting an outer channel or signal route with a non-existing inner channel or signal route.

To remove the error:

- Change the name of the second connection point to match an existing inner signal route or channel.
- Add an inner signal route or channel with the correct name.

(Z.100: 2.5.3, 3.2.2)

ERROR 265 One of the endpoints of path %1 must be the environment

A path that occurs in the second part (i.e. the list of subpaths) of a connection must have the environment as one of its endpoints.

You get this error if:

- You use a signal route or a channel in a CONNECT statement, and the signal route or channel is not connected to the environment.

To remove the error, either:

- Check that the CONNECT statement is connecting the correct signal route or channel in this diagram.
- Change the signal route or channel in this diagram to go to the environment with one of its endpoints.
- Remove the connect statement.

(Z.100: 2.5.3, 3.2.2)

ERROR 266 System definition must contain at least one block or process definition

An SDL system must contain at least one block or process.

You get this error if:

- You have constructed an SDL system that does not contain any blocks or processes.

To remove the error, either:

- Add at least one block with one or several processes to your SDL system.
- Instantiate any existing block type diagrams.

(Z.100: 2.4.2)

WARNING 267 Optional parameter of sort %1 omitted

WARNING 268 Gate must be connected

ERROR 269 Expression evaluates to error

ERROR 270 Selected definition not allowed here

ERROR 271 All packages must either be before or after the system

The system should follow the package list according to SDL. For backwards compatibility the analyzer also accepts a package list after the system, but not both at the same time. In the Organizer this means that the icons for all packages used by a system should be above the system icon.

WARNING 272 No analysis is performed when the infile is empty

The SDL/PR input file cannot be empty.

WARNING 273 No semantic analysis is performed when the input is not a system or a package

Semantic analysis can only be performed on a system or a package.

ERROR 274 Compile and link exited with error code

The code generated by the SDL to C Compiler caused compile or link errors.

ERROR 275 Lost license

The connection to the license server is lost. You must wait until the Analyzer has regained access to the license server before resuming your work. If required, contact your system manager to get the problem solved.

ERROR 276 Cannot get license

No license was available when starting up Analyzer. You must wait until a license becomes available (which will occur when another user terminates his Analyzer).

ERROR 277 Cannot return license

The Analyzer could not return its license to the license pool. To return the license, you may need to stop and restart the license server.

INFO 278 Analyzer command could not be fully performed

The Analyzer did not perform all the passes that were ordered. The results from the Analyzer may not be what you expect. This message is output as a result from a syntactic or semantic error that in turn causes the remaining passes not to be executed.

ERROR 279 The number of block instances must be one or greater

The number of block instances in a multiple block instantiation must be greater than 0.

You get this error if:

- You instantiate a block type in a block reference symbol with a number of instances smaller than one.

To remove the error:

- Change the number of instances to one or more.

ERROR 280 Unreachable path %1 in VIA

The path %1 must be connected to the process instance.

You get this error if:

- You send a signal from a process and the static structure of the SDL system does not declare that you are allowed to send that signal the way you want.

To remove the error, either:

- Add signal routes and channels as needed (with your signal in the signal lists), to reach the wanted process or the environment.
- Remove the sending of that particular signal.

ERROR 281 Unreachable process instance set %1 in TO expr

There must be a path leading to the process instance set %1 from this process instance.

You get this error if:

- You send a signal from a process and the static structure of the SDL system does not declare that you are allowed to send that signal the way you want.

To remove the error, either:

- Add signal routes and channels as needed (with your signal in the signal lists), to reach the wanted process or the environment.
- Remove the sending of that particular signal.

WARNING 282 Several possible paths in VIA list

There are several paths that can convey the signal, one of them will be non deterministically chosen.

WARNING 283 Several possible paths

There are several paths that can convey the signal, one of the paths will be non deterministically chosen during execution.

WARNING 284 Set-Case-Sensitive should be the first command to the analyzer

ERROR 285 Hex string literal must have an even length

This applies to values for types that are strings of bytes.

ERROR 286 Bit string literal must have a length modulo 8

This applies to values for types that are strings of bytes.

ERROR 287 Literal must be one octet long

This applies to values for types that are one byte.

WARNING 288 No matching answer for %2

If a decision (or transition option) is executed with a question not covered by any answer it will be an error. (Z.100:2.7.5 and 4.3.4)

ERROR 289 Expression in transition option evaluates to error

One of the expressions in a transition option is erroneous.
(Z.100:4.3.4)

ERROR 290 External synonym not allowed

External synonyms are only allowed in a number of instances in process reference symbols and in heading symbols in process diagrams.

You get this error if:

- You use an external synonym in the condition for a transition option alternative.

To remove the error:

- Remove the external synonym from the condition for the transition option alternative in question.

ERROR 291 Range check failed

You get this error if:

- You specify a value not allowed for a syntype, like a negative value in a variable of sort Natural.

To remove the error, either:

- Change the value to one of the allowed ones.
- Change the syntype to allow the value.
- Use a different sort allowing the value.

(Z.100:5.3.1.9.1)

ERROR 292 Maximal number of instances equals zero

A process should have at least one instance.

You get this error if:

- You specify zero as the maximum number of instances.

To remove the error, either:

- Change the maximum number of instances to one or more.
- Remove the diagram with maximum number of instances set to zero.

(Z.100:2.4.4)

ERROR 293 Maximum number of instances is less than initial number

In the process instances definitions, the maximum number of instances must be greater or equal to the initial number.

You get this error if:

- You specify a maximum number of instances that is less than the initial number of instances.

To remove the error, either:

- Increase the maximum number of instances to at least the initial number of instances.
- Decrease the initial number of instances to less than or equal to the maximum number of instances.

(Z.100:2.4.4)

ERROR 294 Not allowed to change remote procedure in redefinition

(Z.100:2.4.6)

WARNING 295 Consider adding else answer

If a decision (or transition option) is executed with a question not covered by any answer it will be an error. (Z.100:2.7.5 and 4.3.4)

ERROR 296 Redefined or finalized procedure must be exported when base is

(Z.100:2.4.6)

INFO 297 Error limit reached, terminating

The number of diagnostics that the Analyzer has reported exceeds the value of the limit.

WARNING 298 Accumulated expression depth: %1 parts: %2 matches: %3

Semantic analysis of complex expression may cause long execution time. The execution time will increase in a fashion that is exponential rather than linear. If possible, try to reduce the complexity of expressions by breaking them down into multiple expressions.

WARNING 299 Expression depth: %1 parts: %2

See WARNING 298 Accumulated expression depth: %1 parts: %2 matches: %3 above.

ERROR 300 Command not found

The command that was input to the Analyzer's command interpreter was not recognized.

ERROR 301 Unexpected end of command: %1

The command that was input to the Analyzer does not contain all necessary parameters.

ERROR 302 Ambiguous command

The command that was input to the Analyzer's command interpreter was ambiguous. More characters need to be supplied to identify the command.

ERROR 303 Parameter expected

The command that was input to the Analyzer's command interpreter requires one or more parameters.

ERROR 304 Macro must not be part of a qualifier

A macro is not allowed to be part of a qualifier.

ERROR 305 Remote entity may only be exported once in a process

This applies to exported procedures and variables.

(Z.100:2.4.6 and 2.6.1.1)

ERROR 306 No input specified

There is no input set to the Analyzer using the set-input command.

WARNING 307 %1: unknown, line ignored!

The command line is ignored.

ERROR 308 More than one package match use statement

Use Modules to group systems together with their package lists to make package names unique within a module.

INFO 309 Location of the shadowed definition

ERROR 310 Selected item %1 is not connected to a file

The Organizer has ordered to analyze a diagram that is not connected to a file.

ERROR 311 Signal %1 found in opposite direction in connection

You get this error if:

- You connect two signal routes or channels where a signal is defined according to the signal lists to go in different directions on the inside and on the outside of the connection.

To remove the error:

- Reverse the direction of one of the signal routes or channels, at least for the signal that caused the error.

ERROR 312 Syntax error in rule %1, symbol %2 found but one of the following expected:

Syntax error!

ERROR 313 Syntax error, symbol %1 found but one of the following expected:

Syntax error!

ERROR 314 Lexical error in rule %1, symbol %2 found but one of the following expected:

Syntax error!

You get this error either if:

- You have typed a character or keyword that is not allowed in a CHARSTRINGLITERAL in a place where the syntax checker thinks there should be a CHARSTRINGLITERAL.
- You have used an SDL keyword as a name for a variable.

To remove the error in the above cases, either:

- Examine the pointed out character or keyword to see if it really belongs there.

- Try to find out why the syntax checker expects a CHARSTRING-LITERAL; did you intend to have a CHARSTRINGLITERAL there?.
- Rename any variable that you have given the same name as an SDL keyword.

You also get this error if:

- You have constructed a word or token out of single characters that does not match allowed word or tokens in SDL.

To remove the error in the above case:

- Check the word in question and replace any mistyped or misplaced characters.

ERROR 315 Lexical error, symbol %1 found but one of the following expected:

Syntax error!

ERROR 316 File %1 not found in environment variable %2 %3, check installation!

Check the installation. If the error persists, contact IBM Rational Customer Support. Contact information for IBM Rational Customer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*.

ERROR 317 Outside connect item %1 is locally defined

You get this error if:

- Your connect statement mentions an inner channel or signal route instead of a channel or signal route outside the current diagram as the first connect item.

To correct the error, either:

- Swap places (if the second connect item is wrong too) on the inner and outer channel or signal route in the connect statement.
- If there is a name conflict with a locally defined item, then add a qualifier to the outer connect item.

WARNING 318 %1 : signal not used
The signal %1 should be used.

Here is a small explanation valid for all *<entity> not used* warning messages:

You get this error if:

- You declare an entity but do not use it.

To remove the error, either:

- Remove the declaration.
- Start using the entity.
- Rename the entity in the declaration to match a used entity of the same type that does not have a declaration.
- Change the type in the declaration to match a used entity with the same name but with a different type (that does not have a declaration).

WARNING 319 %1 : timer not used in input
The timer %1 should occur in an input.

WARNING 320 %1 : timer not used in set
The timer %1 should be set in the process.

WARNING 321 %1 : signallist not used
The signallist %1 should be used.

WARNING 322 %1 : variable not used
The variable %1 should be used.

WARNING 323 %1 : formal parameter not used
The formal parameter %1 should be used.

WARNING 324 %1 : synonym not used
The synonym %1 should be used.

WARNING 325 %1 : viewed variable not used
The viewed variable %1 should be used.

WARNING 326 %1 : remote variable not used
The remote variable %1 should be used.

WARNING 327 %1 : sort not used
The sort %1 should be used.

WARNING 328 %1 : generator not used
The generator %1 should be used.

WARNING 329 %1 : label not used
The label %1 should be used.

WARNING 330 %1 : system type not used
The system type %1 should be used.

WARNING 331 %1 : block type not used
The block type %1 should be used.

WARNING 332 %1 : process type not used
The process type %1 should be used.

WARNING 333 %1 : procedure not used
The procedure %1 should be used.

WARNING 334 %1 : package not used
The package %1 should be used.

ERROR 335
Reserved for future purpose.

ERROR 336 Input to Instance generator must be a system

ERROR 337 Operator diagram/definition not allowed where constant required

You get this error if:

- You call an operator in a place where a constant is required, for instance in a declaration of a variable.

To remove the error, either:

- Remove the operator call, and replace it with a constant if needed.
- Call a non operator definition operator.

WARNING 338 %1 : operator diagram/definition not used

WARNING 339 %1 : operator not used

WARNING 340 %1 : remote procedure not used

ERROR 341 No matching variable, formal parameter, synonym or literal

ERROR 342 Literal %1 in literal renaming is not defined in parent sort

ERROR 343 No matching field selection, array element access or prefix operator

ERROR 344 No matching quoted operator

You get this error if:

- You try to use a quoted operator with the wrong number of arguments or the wrong type of arguments.

To remove the error, either:

- Check that you are using two arguments for infix operators and one argument for monadic operators.
- Check that the operators you are using are of the correct type.

ERROR 345 No matching (. .)

ERROR 346 No visible variable or formal parameter with this name

ERROR 347 Type mismatch in component selection of variable or formal parameter

ERROR 348 No matching monadic operator

ERROR 349 No matching infix operator

INFO 350 %1 is one of the possible matching types

ERROR 351 No matching remote variable in variable definition

INFO 352 %1 is one of the visible definitions

WARNING 353 Type mismatch between actual and formal parameter

The types of the actual and formal parameters are implemented with different typedef, i.e. they may be of different size. Only checked for functions that cannot handle length mismatch.

ERROR 354

Reserved for future purpose.

ERROR 355 No matching remote variable in import definition

ERROR 356

Reserved for future purpose.

ERROR 357 No matching character string

ERROR 358 No matching choice primary

ERROR 359 Now not allowed where constant required

ERROR 360 No matching now expression

ERROR 361 Import expression not allowed where constant required

You get this error if:

- You use an import expression where a constant is required.

To remove the error:

- Replace the import expression with a constant.

ERROR 362 PId expression not allowed where constant required

You get this error if:

- You are using a pid expression where a constant is expected.

To remove the error:

- Replace the pid expression with a constant expression.

ERROR 363 No matching PId expression

You get this error if:

- You use a PId expression where a boolean value is expected.

To remove the error, either:

- Replace the PId expression with a boolean expression.
- Make the expected value to be of type PId.

ERROR 364 View expression not allowed where constant required

You get this error if:

- You are using a view expression where a constant expression is expected.

To remove the error:

- Replace the view expression with a constant expression.

ERROR 365 Timer active expression not allowed where constant required

ERROR 366 Any expression not allowed where constant required

ERROR 367 No matching dereferencing operator

ERROR 368 Data base error

An internal error occurred in the Analyzer. Please send a report to IBM Rational Customer Support. Contact information for IBM Rational Cus-

tomer Support can be found in "How to Contact Customer Support" on page *iv in the Release Guide*.

ERROR 369 No matching field selection

You get this error if:

- You try to access a field in a struct using a name that does not represent any field in the struct.
- You try to access a field in a struct whose sort does not match the one expected.

To remove the error:

- Make sure you are using the correct name to access a field in the struct.
- Add a field in the struct with the wanted name.
- If required, reference a conversion operator to have the field match the sort required.

ERROR 370 Type mismatch for procedure return value

ERROR 371 Type mismatch for variable, formal parameter, synonym or literal

You get this error if:

- You assign a variable a value of the wrong type.

To remove the error:

- Assign the value to a variable of the correct type instead.

ERROR 372 No visible variable, formal parameter, synonym, literal or operator with this name

You get this error if:

- You use the name of a variable, formal parameter, synonym, literal or operator that is not defined in the current scope.

To remove the error, either:

- Change the name to match a defined entity.
- Define the entity.

ERROR 373 No visible synonym or literal with this name

You get this error if:

- You refer to a synonym or a literal that cannot be seen from the current scope.

To remove the error, either:

- Correct the spelling mistake, if it was not meant to be a synonym or literal.
- Add a synonym or literal with the given name.

ERROR 374 Variable or formal parameter not allowed where constant required

You get this error if:

- You are using a variable or formal parameter in a place where a constant expression is expected.

To remove the error:

- Replace the variable or formal parameter with a constant expression.

ERROR 375 Type mismatch for synonym or literal

You get this error if:

- You use a boolean synonym or literal where an integer synonym or literal is expected.

To remove the error:

- Replace the boolean with an integer.

ERROR 376 Only one reference to each diagram allowed

Several diagrams connected to the same file.

ERROR 377 Keyword call required for procedure call

You get this error if:

- You are calling a value returning procedure without using the CALL keyword.

To remove the error:

- Add the CALL keyword before the procedure name in the expression.

ERROR 378 Several macrodefinitions have the same name %1

The name of a macro must be unique.

WARNING 379 Macrodefinition %1 was never called

The macro %1 should be used.

ERROR 380 Names separated by %, %1%%2, where neither is a formal parameter nor MACROID in macrodefinition %3

(Z.100: 4.2.1)

WARNING 381 Formal parameter %1 is never used in macrodefinition %2

The formal parameter %1 should be used in the macro definition

ERROR 382 The macrodefinition name is %1, but the endmacro name is %2

The names in the macro definition and the endmacro must be equal.

ERROR 383 Two formal parameters with the same name %1 in macrodefinition %2

Two formal parameters of a macro cannot have the same name.

ERROR 384 No matching macrodefinition found

The corresponding macro definition is missing.

ERROR 385 A macro may not call itself

A macro definition cannot be circular.

WARNING 386 No macro found

The system does not contain any macros or calls.

ERROR 387 Start transition expected

A process or procedure definition must begin with a start transition.

You get this error if:

- You define a process without defining a start transition.

To remove the error:

- Add a start transition to the process.

ERROR 388 Illegal redeclaration of operator definition

You get this error if:

- You declare an operator twice.

To remove the error, either:

- Rename one of the operators if they are different and you need them both.
- Remove one of the operator definitions.

(Z.100: 5.3.2)

ERROR 389 No matching operator signature found

The operator must be defined for this signature.

ERROR 390 This unexpected

The keyword this is only allowed in a procedure call, process create or “output to”.

You get this error if:

- You use the keyword THIS in normal diagrams.

To remove the error, either

- Remove or replace the keyword THIS.
- Convert the diagram to a type-based diagram.

ERROR 391 Context parameter %1 not allowed

(Z.100: 6.2)

ERROR 392 Call this invalidated by adding parameters to sub type

Call this cannot be used when formal parameters are added in the sub-type.

You get this error if:

- You use the keyword THIS in a procedure call and the called procedure is a super type to a sub type that adds parameters to be used in a procedure call.

To remove the error, either:

- Make both the sub type and the super type use the same set of parameters by declaring all parameters in the super type if you intend to call the sub type.
- Remove the THIS keyword if you intend not to call the sub type.

INFO 393 Instance path to the problem in previous message

This message provides additional information that may be used to locate the source of error in the previous message. This message may appear repeated times, depending on the depth of the SDL structure where the previous error is detected.

ERROR 394 Package Predefined expected, check installation

File *predef.sdl* does not contain the package Predefined. The error message appears only when you have altered the installation in an improper way.

ERROR 395 Type mismatch for view expression

You get this error if:

- You view a variable of the wrong type. For instance, you assign a variable of type boolean the value of a viewed variable of type integer.

To remove the error, either:

- Change the type either of the assigned variable or the viewed variable to match each other.
- View another variable instead, that has the correct type.
- Assign to another variable instead, that has the correct type.

ERROR 396 Viewed variable %1 is not revealed in block %2

The viewed variable must be declared as revealed in block %2.

ERROR 397 Context parameters are not supported

Context parameters are not implemented. Consider using an alternative approach.

WARNING 398 One line external formalism name expected on same line as alternative keyword

Syntax error! When using alternative the name of the alternative data formalism must occur on the same line as the alternative keyword.

ERROR 399 Type mismatch for timer active expression

You get this error if:

- You are using the result of the ACTIVE call as if it had another type than boolean.

To remove the error:

- Make sure you are using the result of the ACTIVE call as a boolean value.

ERROR 400 Type mismatch for any expression

Chapter

55 *Analyzing a System*

Analyzing a system means checking that its SDL description obeys the syntax and semantic rules as defined in the Z.100 recommendation (some syntax checking is performed by the SDL Editor at editing time).

The SDL Analyzer allows you to perform a complete syntactic and semantic check of an SDL system. This chapter describes the Analyzer and how you may use it to analyze an SDL system.

For a reference to the Analyzer commands and the Analyzer functionality and restrictions, see *chapter 54, The SDL Analyzer*.

General Description

The Analyzer's main task is to perform syntactic and semantic analysis of SDL-92 definitions and diagrams, and to serve as a front end to code generators. You may perform full syntactic and nearly full semantic analysis of complete system definitions.

Analysis of separate units (block, process, substructure, service, and procedure) is also supported. Syntactic analysis may be performed on a unit, while restricted semantic analysis of a unit may only be performed if the context of the unit is provided. The context is the enclosing units and their definitions (for a detailed description, see the section *"Separate Analysis" on page 2505 in chapter 54, The SDL Analyzer*).

The Analyzer works in a number of passes:

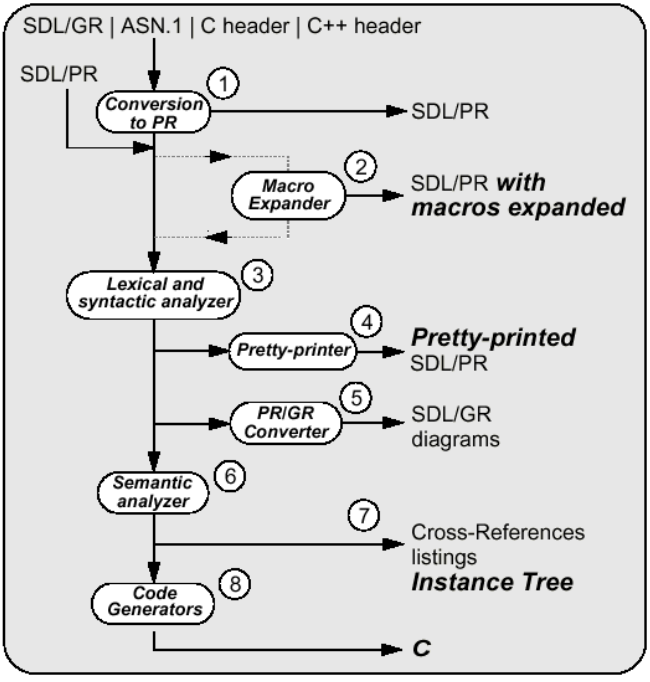


Figure 493: Analyzer passes

1. *Conversion to PR*. This pass is needed when input is not SDL/PR, i.e. for SDL/GR diagrams, ASN.1 files, C, and C++ header files. The result is a PR file with a "raw" layout, that is submitted as input to the next pass. This PR file is not intended to be read by the human. See *"Conversion to PR" on page 2501 in chapter 54, The SDL Analyzer* for a more thorough description.

2. *Macro expansion.* This pass is needed when input contains references to macros (which need to be expanded). The result is a PR file where SDL macros are expanded. See *"The Macro Expander" on page 2502 in chapter 54, The SDL Analyzer* for a more complete description of the Macro Expander.

3. *Lexical and syntactic analysis.* This pass checks that the input follows the SDL definition with respect to syntactic rules. During this pass, the Analyzer builds an abstract syntax tree that is used by the following passes. See *"The Lexical and Syntactic Analyzer" on page 2504 in chapter 54, The SDL Analyzer* for a more thorough description.

4. *Pretty-printing*. This optional pass uses the pretty-printer to produce an SDL/PR file with a nice layout, easy to read and understand by the human. The pretty-printed PR file contains the original SDL specification formatted according to specific layout rules.

5. *PR to GR conversion.* The input PR files are translated to SDL/GR diagrams, that you may open in the SDL Editor. This pass allows you to for instance import PR files from other tools supporting SDL. See also section *"The PR to GR Converter" on page 2507 in chapter 54, The SDL Analyzer*.

6. *Semantic analysis.* During this pass, the Analyzer checks that your SDL diagrams obey the static semantic rules as defined in the Z.100 recommendation. During this pass, the Analyzer builds and uses a symbol table, which can also be used later by the Code Generators. The semantic analysis is likely to be the "bottleneck" when analyzing a system. See "Optimizing a System to Reduce Analysis Time" on page 2621.

7. After the semantic analysis pass, you may optionally generate:
   - *cross-references* listings of SDL entities; where they are defined in an SDL system, and where they are used (referred). The result from this pass is a file which contents may be displayed graphically in the Index Viewer. See chapter 46, *The SDL Index Viewer* for more information.
   - *instance tree* information about the SDL system. The result from this pass is an instance information file consisting of records that describe the SDL entities that are present in the system after instantiation of all types. The file is a plain text file with a simple format. See *"File Syntax" on page 2513 in chapter 54, The SDL Analyzer* for more information.

8. If your configuration includes a Code Generator[1] you may include:
   - A *C Code Generation* pass, in order to generate a C description of your SDL system. This C code is then compiled and linked to generate a simulator, an explorer or an application. The SDL to C Compiler is available as a Cbasic and as a Cadvanced code generator.
   - A *Cmicro Code Generation* pass. The generated C code is optimized with respect to memory requirements, making it suitable for generating applications for systems with limited resources.

Analyzer Input and Output

The input to the Analyzer consists of SDL-92 specifications, that is, SDL/GR diagrams that are stored using the storage format of the SDL Suite, or SDL/PR files, or a combination of both.

The output consists mainly of PR files, error and warning messages. These messages are presented on the screen in a log window and may be stored on file. See "Error and Warning Messages" on page 2531 in chapter 54, *The SDL Analyzer* for a description of these messages.

It is also possible to obtain a pretty printed SDL/PR file of the input and to transform SDL/PR files into SDL/GR diagrams.

1. Although technically built into the Analyzer binary executable (sdtsan), the Code Generators are additional optional tools that are licensed separately.

The Analyzer User Interfaces

The Analyzer provides the following user interfaces.

Graphical UI

When started from the Organizer with the *Analyze* command, the Analyzer takes advantage of the graphical user interface and integration mechanism of the SDL Suite.

For instance:

- Graphical references between source documents and error reports is supported, which facilitates locating and correcting errors in the source SDL diagrams.
- An SDL-Make facility, managed by the Organizer, controls the Analyzer and brings down the analysis work that needs to be done.
- On-line help on analysis diagnostics is available (provided the prerequisites for the on-line help are satisfied).

Batch UI

Suitable for running the Analyzer non-interactively. See "Batch Facilities" on page 208 in chapter 2, *The Organizer*.

Command-Line UI

Suitable for working on the file level with detailed control. See "The Analyzer Command Line UI" on page 2474 in chapter 54, *The SDL Analyzer*.

Using the Analyzer

This section describes how you operate the Analyzer from the Organizer. We will discuss topics related to the various ways you may analyze an SDL structure. With the SDL Suite, you may for instance perform syntax check on an SDL structure or check an entire SDL system with respect to the semantic rules.

Analyzing Using Default Options

To analyze an SDL structure using default options:

1. In the Organizer, select the root node for the subtree that is the subject to be input to the Analyzer.

2. Click the quick button for Analyze. The Organizer first checks if there are any unsaved diagrams; if any, the Organizer will prompt you to save these before analyzing them (since the Analyzer operates on the latest saved copy of a diagram).

3. The Organizer determines what diagrams need to be analyzed and what passes need to be run, by looking at the time the diagrams were saved on file and by monitoring the Analyzer's work.

   – To perform an analysis, you may either "touch" the SDL diagram files or force the Analyzer by clicking the *Full Analyze* button (see Figure 494).

4. The analysis job is submitted the to the Analyzer, using the options as they are currently defined in the Analyzer options dialog (see Figure 494 on page 2619).

5. From now on, the status bar reads "Analyzer working". When done, the status bar will read "Analyzer ready".

Analyzing Using Customized Options

To analyze an SDL structure with customized options:

1. In the Organizer, select the root node for the subtree that is the subject to be input to the Analyzer.

2. From the Organizer's *Generate* menu, select the *Analyze* command. The Analyzer Options dialog is displayed.

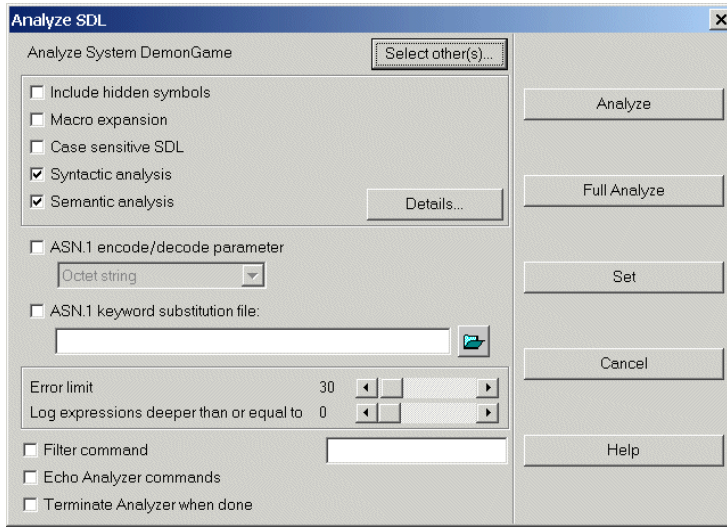


Figure 494: The Analyzer dialog

3. Adjust, if required, some of the Analyzer options to their required values. These options govern what passes should be performed by the Analyzer, see Figure 493 on page 2614, and what output should be produced by the Analyzer. (The code generation pass options are however not controlled from this dialog.)

4. Click the *Analyze* button. First, the Organizer checks if there are any unsaved diagrams; if any, the Organizer will prompt you to save these before analyzing them (since the Analyzer operates on the latest saved copy of a diagram).

5. The Organizer determines what diagrams need to be analyzed and what passes need to be run, by looking at the time the diagrams were saved on file and by monitoring the Analyzer's work.

   – To perform an analysis, you may either "touch" the SDL diagram files or force the Analyzer by clicking the *Full Analyze* button.

6. The Analyzer is initialized and starts executing (this is indicated in the Organizer status bar which now reads "Analyzer working").

7. When the Analyzer has analyzed the input as specified in the options, the Organizer status bar reads “Analyzer ready”. The results of the analysis are reported in the Organizer log window.

Performing Syntax Check

1. If the input contains SDL macros, you should expand these before proceeding with the syntax analysis in the resulting SDL/PR file.
   – Turn the *Macro expansion* button on to order expansion of SDL macros.
2. Turn the *Syntactic analysis* button on to include the syntax checking pass.
3. Click the *Analyze* button.

Performing Semantic Check

Turn the *Semantic analysis* button on to include the semantic check pass. This option cannot be turned on unless the *Syntactic analysis* pass is enabled. The semantic checker may be set up with by a number of options, each one individually activated by a toggle button.

Note: Optimizing a System to Reduce Analysis Time

There are two components that, to a large extent, affect the performance of the "resolution by context" in the semantic analysis pass:

- The depth of the expressions, because all possible combinations must be tried.
- The size of the system, because the context is all the visible identifiers.

Deeply nested expressions may cause a **significant** degradation of performance when performing the semantic analysis pass. It is therefore recommended to break down complex expressions into multiple, less complex expressions. *Checking for Deep Expressions* (see below) can assist you doing this.

If it is an option to modify your system, it might be worthwhile to go through all synonyms, newtypes and syntypes at the system level and move them as far as possible down the system structure. This makes the context visible to every expression smaller and reduces the time spent in "resolution by context". (Doing the same thing at lower levels will also improve performance, but not as much as on higher levels.) The Index Viewer might be useful in accomplishing this.

Checking Output Semantics

The *Check output semantics* option controls whether to issue warnings when SDL signal sendings, where the semantics is different in SDL-88 and in SDL-92, are detected. These warnings are particularly valuable when the input consists of SDL diagram that were designed in SDL-88 (for instance with SDT 2.X).

Detecting Not Used Definitions

The *Check unused definitions* option, when turned on, orders the Analyzer to report definitions that are not used (for instance variables that are declared but neither written or read).

Checking Optional Parameters

The *Check optional parameters* option controls whether to issue warnings when an optional parameter is omitted from a procedure call, an output, a create request, or an input. Note that in/out parameters are not

optional. An omitted parameter is indicated by empty parenthesis or a comma. See also *Checking Trailing Parameters*, below.

Checking Trailing Parameters

The *Check trailing parameters* option controls whether to issue warnings when the number of actual parameters is not equal to the number of formal parameters in a procedure call, an output, a create request, or an input. See also *Checking Optional Parameters*, above.

Checking Unique References

When the *Check references* option is activated, the Analyzer will check that each remote definition is referred only once. Turn this button off to disable this check.

Checking Missing Else Answer

The *Check missing else answers* option controls whether to issue warnings when an else part is expected but does not exist among the branches in a decision or transition option statement.

Checking Missing Answer Values

The *Check missing answer values* option controls whether to issue warnings when there are values not covered by any of the branches in a decision or transition option statement.

Checking Parameter Mismatch

The *Check parameter mismatch* option controls whether to issue warnings when the types of the actual and formal parameters are implemented with different typedef, i.e. they may be of different size. This is only checked for functions that cannot handle length mismatch.

Externel types should call GenericFree

The *External types should call GenericFree* option controls whether to generate code to call GenericFree for variables of externally defined types. Note this is a codegenerator option.

Allowing Implicit Type Conversions

The option *Allow implicit type conversion* controls whether implicit type conversions of reference data types (generators Own, ORef, and

Ref) are allowed. For more information, see "Implicit Type Conversions" on page 134 in chapter 3, *Using SDL Extensions, in the SDL Suite 6.2 Methodology Guidelines*.

> **Note:**
> Analyzing large expressions with this option on is slow.

Generating a Cross Reference File

Turn the *Generate a cross reference file* option on to have the Analyzer generate a file with a list of definitions and references to SDL entities, as an supplementary result from the semantic pass. You may also want to specify another file name than the suggested one. The contents of this file may be read and visualized graphically with the Index Viewer.

Generating a Complexity Measurement file

Turn the *Generate a complexity measurement file* option on to have the Analyzer generate a file containing characteristics of the system. You may change the file name in the field below. See chapter 48, *Complexity Measurements* for more information.

Generating an Instance Information File

Turn the *Generate an instance information file* on to have the Analyzer generate a file with instance information about the analyzed system. The name of the file can be set in the field below. The file is produced after the semantic pass. Read more about the contents of an instance information file in *"SDL Instance Information" on page 2512 in chapter 54, The SDL Analyzer*.

Checking for Deep Expressions

Adjust the *Expression depth* parameter to specify the depth limit that the Analyzer should check for when evaluating expressions. Expressions which depth exceed the specified limit will be reported. Where possible, try to break down deep expressions since they require advanced calculations and slow down the Analyzer.

Specifying the Error Limit

Adjust the *Error limit* parameter to an adequate value (drag the slider for coarse adjustments, click left or right on the bar for fine adjust-

ments). The Analyzer will stop its execution once this error limit has been reached.

Using a Filter

With the *Filter command* files can be filtered or preprocessed before they are read by the Analyzer. The specified executable file will be called with two arguments: the file to be processed by the Analyzer, and the Analyzer pass to be executed (import, macro, or parse). Try a simple OS command and look in the Organizer log to find out exactly how it is called.

Tracing the Analyzer Execution

You can trace the execution of the Analyzer by turning on the option *Echo Analyzer commands*. All Analyzer commands are then printed in the Organizer Log as they are executed.

Terminating the Analyzer Process

Turn on the option *Terminate Analyzer when done* if you want the Analyzer process to terminate after analysis is done. Otherwise, the Analyzer process is left running in the background.

Locating and Correcting Analysis Errors

The results of the Analyzer are appended to the Organizer Log Window. (You may save the window contents on any file at any time). The results of the last run are also saved on a file with the predefined name `analyzer.err`

The SDL Suite provides a nice feature for displaying the source of an analysis error:

1. Locate the Organizer log (select *Organizer Log* if required).
2. Select the error (or warning) message by dragging the mouse.

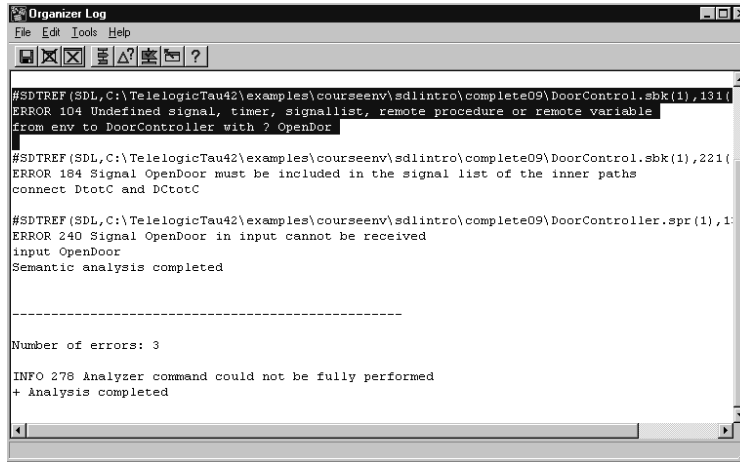


Figure 495: Selecting an error message

– The selection must contain at least the line starting with the text #SDTREF (for more information on the format of references, see chapter 18, *SDT References*).

3. Select the menu choice *Show Error* from the *Tools* menu.

4. The symbol where the error has been detected is displayed in an SDL Editor window. When the reference also contains a line and a column reference, the text cursor is placed on the line and column of text where the error was detected.

5. If you need additional explanations to understand the error message, select *Help on Error* from the *Tools* menu.

Producing a Pretty-Printed SDL/PR File

You may produce a pretty-printed PR file with the Analyzer, with a single command. The layout used by this PR makes it easy to read by the human. You may either submit SDL diagrams or SDL/PR files as input to the pretty-printer.

To be able to pretty-print SDL/PR, the SDL Suite requires that the input must be syntactically correct. If the input does not fulfill this requirement, the SDL Suite will however produce a PR file with a “raw” layout. This file is used by the SDL Suite as a temporary file and it does not address the human reader.

1. In the Organizer, select the root node of the SDL structure to be pretty-printed.
2. From the Generate menu, select *Convert to PR/MP*. The *Convert to PR* dialog is displayed:

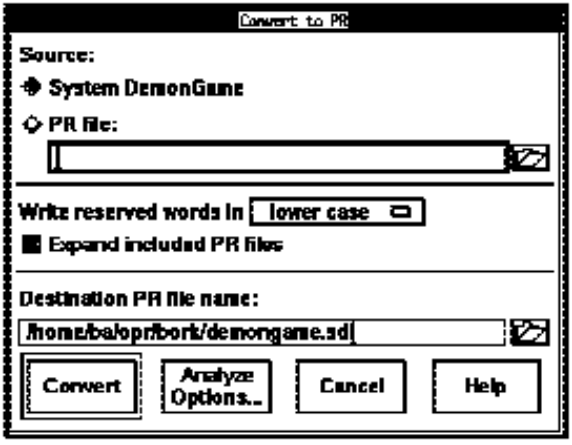


Figure 496: The Convert to PR dialog

3. Adjust, if required, the conversion options:
   – You may specify a PR file as alternative input. (The default input is the SDL structure designated by the object selected in the

Organizer.) To do this, type in the name of a file into the *PR file* text field.

- You may order the Analyzer to capitalize SDL reserved words or not; select the appropriate option from the *Write reserved words in* option menu.
- If the SDL structure contains #INCLUDE references to SDL/PR files, you may want to expand these and include them in the resulting pretty-printed PR file. Turn the *Expand included PR files* button on to order expansion of included PR files.

4. An output file where the resulting SDL/PR code will be stored is suggested. You may specify any other file to store the results on in the *Destination PR file name* text field.
5. If the input contains SDL macros, you may want to expand these in the resulting SDL/PR file.
   - Click the *Analyze Options* button to gain access to the *Analyzer options* dialog, where you turn the *Macro expansion* button on and click the *Set* button.
6. Click the *Convert* button to close the dialog and have the Analyzer start the conversion to SDL/PR.

Converting SDL/PR to SDL/GR

You may convert SDL/PR files to SDL/GR diagrams, storing them on files using the native format of the SDL Suite. Once converted, these SDL descriptions may be managed, edited and printed by the graphical tools of the SDL Suite just as if they were created using the SDL Suite.

To convert an SDL/PR file to SDL/GR diagrams:

1. From the Organizer, *Generate* menu, select *Convert to GR*. A dialog is displayed:

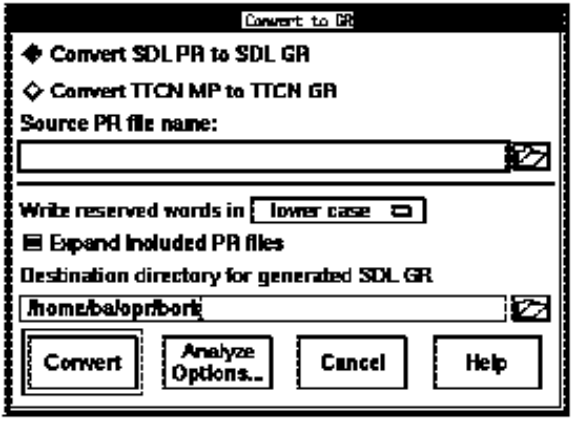


Figure 497: The Convert PR to GR dialog

2. Specify the PR file to convert from. You may either type the file specifier or click the folder button to issue a File Selection Dialog.

3. Specify the destination directory, where to store the results of the conversion. You may either type the name of a directory or click the folder button to issue a directory selection dialog.

4. You may specify other options:

   – Whether to capitalize SDL reserved words or not, by selecting the corresponding option from the option menu.

   – Whether to expand any included PR files or not (PR files are included using a #INCLUDE directive inside an SDL comment statement).

5. If required, click the *Analyze Options* button to modify the Analyze options (typically, you may need to turn the *Macro Expansion* option on). Close the *Analyze Options dialog* with the Set button (see Figure 494 on page 2619).

6. Click the *Convert* button to order the conversion. The Organizer's status bar reads "Analyzer ready" when the conversion is completed.

   – Each diagram that is specified (or included and expanded) in the source PR file is transformed to one file. These files are assigned default file names (see "Save in file" on page 63 in chapter 2, *The Organizer*).

   – The generated diagrams are assigned a layout by the SDL Editor.

   – Error and warning messages, if any, are appended to the Organizer log window.

7. To look at the resulting diagrams, you may either open the resulting files in the SDL Editor.

   – Alternatively, identify the root diagram file that has been produced. (You may need to look at the input SDL/PR file to determine what file to look for.) Then, import this file into the Organizer, with *Import SDL* from the *File* menu. Specify the root diagram file as *Root document*.

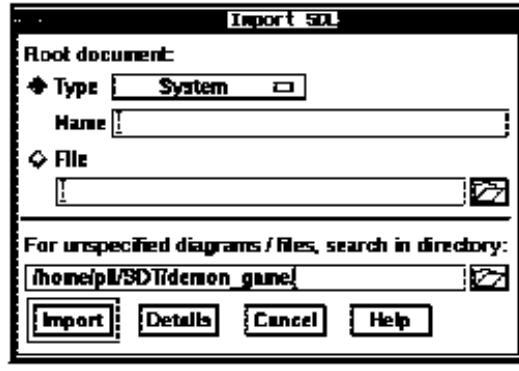


Figure 498: Importing an SDL structure

Chapter 55 Analyzing a System

Chapter

56

The Cadvanced/Cbasic SDL to C Compiler

The Cadvanced/Cbasic SDL to C Compiler translates your SDL system into a C program that you can compile and link together with a runtime library to form an executable program such as a simulator, an explorer or, in the case of Cadvanced, an application.

This chapter is a reference manual to the Cadvanced/Cbasic SDL to C Compiler. There are also a number of other chapters related to code generation:

- **In *chapter 49, The SDL Simulator*, you will find a reference to the simulation facilities in the SDL Suite. In chapter 50, *Simulating a System*, you will find a user's guide to the simulator.**
- **In *chapter 52, The SDL Explorer*, you will find a reference to the validation facilities in the SDL Suite. In chapter 53, *Validating a System*, you will find a user's guide to the SDL Explorer.**
- **In chapter 57, *Building an Application*, you may read about how to generate applications for host and target environments.**
- **In chapter 61, *The Master Library*, you will find information about how to customize your own libraries for a specific purpose, such as application generation for target computers. The chapter also describes the structure of the generated C code and the internal data structures in the generated C code.**
- **In chapter 62, *The ADT Library*, you will find a reference to the library of abstract data types that is distributed with the SDL Suite and that you may use in your systems designed in SDL. Some examples of use are also available.**
- **In chapter 63, *The Performance Library*, you may read about how to generate and run simulators which are specially adapted for the area of performance simulation.**

Introduction

Application Areas for the Cadvanced/Cbasic SDL to C Compiler

There are a number of application areas for the Cadvanced/Cbasic SDL to C Compiler, for example:

- Functional simulation and debugging of protocol specifications
- Debugging of system designs described in SDL
- Generation of applications, including embedded system applications with real time characteristics
- Performance simulations
- Simulation of the behavior behind a user interface prototype

In this part of the manual, the general behavior of the code generator and its application for simulation and debugging are discussed. The possibility to generate simulators is described in chapter 49, *The SDL Simulator*.

Functional Simulation and Debugging

During the validation of a specification or design of an application expressed in ITU SDL, you can use the Cadvanced/Cbasic SDL to C Compiler as a tool for simulation to help you understand and debug the behavior of a system description. (See *chapter 49, The SDL Simulator*.)

Errors arising from two different areas have to be considered in the validation process. In the language domain, errors due to illegal or illogical usage of the language concepts might be introduced into the specification; while in the problem domain, logical errors might be introduced.

With traditional computer program development, most illegal uses of language concepts are found by compilers or by run-time systems. Examples are syntax errors, missing declarations, division by zero, or indexing an array out of its bounds.

In the problem domain, however, the only feasible ways of detecting logical errors in non-trivial programs are testing and proofreading. When it comes to specifications in SDL, language domain errors can be

detected by using the SDL Analyzer, which can be seen as a compiler without a code generation facility (see *chapter 54, The SDL Analyzer*). To detect problem domain errors, testing by simulating the specification is the main procedure available. Please see also chapter 52, *The SDL Explorer*.

The specification of a protocol in SDL, for instance, specifies a signal interface by giving a hypothetical implementation of the components in the protocol. This strategy immediately brings up two different purposes for simulating the behavior of a system specification: to understand the external view and to understand the internal view.

In the external view, the signal interface is of concern, while the internal behavior of the system specification (the behavior of the processes in the system) is of little or no interest. In the internal view, the internal behavior of the system specification is of concern, while the external signal interface is simply seen as part of the internal behavior.

A simulation of the internal behavior of a system specification constitutes an important part of the validation of the specification, both as a debugging tool and as a means to increase the understanding of the dynamic behavior of the specification. A designer of a system might use this kind of simulation to understand the specification better.

The ability to simulate and debug applications generated by the code generator at an SDL level is a very important feature towards achieving the correct overall behavior of the application. The debugging facilities provided by the SDL Suite have much in common with interactive debuggers for ordinary programming languages. The debugging is performed on a host computer.

Another application of the code generator as a simulator generator, is of course in SDL education, where simulation, especially of the internal behavior of a system specification, can serve as a powerful way of clarifying the semantics of SDL concepts.

Performance Simulation

The Cadvanced/Cbasic SDL to C Compiler can be used for performance simulations. You describe the performance model of the actual system using SDL. This model can be translated to a simulation and executed. By introducing measurements of interesting data, such as queue lengths, delays, and so on, into the SDL model, it is possible to gather statistical data during the execution of the simulation. In chapter 63, *The*

Performance Library, you can find a description of the performance simulation facilities.

To simplify this kind of simulation, a number of SDL abstract data types and their implementations have been developed, where, for example, random number generation and handling of queues are supported. Please see chapter 62, *The ADT Library*.

Validation

The SDL Explorer uses the code produced by the Cadvanced/Cbasic SDL to C Compiler to form a program suitable for validation of an SDL system. The Explorer uses state space exploration and can be used to:

- Find run time errors
- Verify MSCs against the SDL system
- Verify user defined rules

The Explorer is described in *chapter 52, The SDL Explorer*.

Communicating Simulations

You can specify that a generated C program should be able to communicate over the *PostMaster*, which is the mechanism used for communication between the SDL Suite tools. Signals sent from the SDL system (the generated program) to the environment and signals coming to the SDL system from the environment can be handled. This facility makes it, for example, possible to develop simulation programs for two communicating systems, execute them using the SDL Suite and obtain communication between the systems.

As a generated C program does not know what it communicates with, it can of course communicate with any type of application, as long as the application is connected to the *PostMaster* (the communication medium) and sends signals according to the defined format. How to achieve this is described in *chapter 12, Using the Public Interface*.

A very interesting group of such applications are user interfaces. By connecting a user interface and an SDL simulation you can achieve several things: You can, for example, build well-designed application oriented user interfaces that present what is going on in a simulation, or you can in a simple way define the logic behavior behind a user interface during its prototyping phase.

Overview of the Cadvanced/Cbasic SDL to C Compiler

To facilitate the validation of SDL specifications or descriptions, the SDL Analyzer contains an SDL parser, an SDL semantic checker, and the Cadvanced/Cbasic SDL to C Compiler.

Creating a C Program

To obtain an executable program that behaves according to an SDL description, you enter the SDL description into the SDL Analyzer, which contains the Cadvanced/Cbasic SDL to C Compiler. If the SDL description is syntactically and semantically correct, a C program is generated. You then compile this program using an ordinary C compiler and link it with a predefined SDL run-time library to form an executable program. See Figure 499.

As indicated above, the C code generation facility contains two components:

- The SDL to C Compiler, which can be seen as a back-end to the SDL Analyzer. This component generates a C program.
- Predefined and precompiled C units, which implement an **SDL runtime library** and the command line user interface of a simulator, that is, a **monitor system**. The run-time library also includes a **communication mechanism** which makes it possible to trace the execution of SDL transitions in the SDL Editor. There are several versions of the library that are suitable to different application areas for the generated C code, see "Libraries" on page 2770 in chapter 57, *Building an Application*.

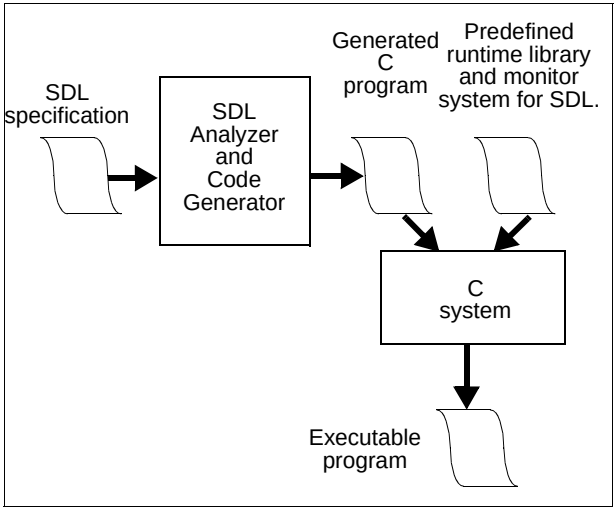


Figure 499: The production of an executable program

Executing a C Program as a Simulator

The generated C program uses an internal data representation of the SDL objects in the system, for example process instances and signal instances. The process instances will execute transitions in a quasi-parallel manner. During a transition, SDL actions such as tasks, decisions and signal outputs are executed according to the semantics of SDL.

You interact, using SDL terminology, with the simulator through a monitor system, which contains a number of commands to:

- Control the execution of transitions.
- Examine the status of objects in the system.
- Turn log facilities on and off.
- Affect the system by, for example, sending signal instances from the environment to the system.

Contents of This Chapter

You can find more details on creating and executing a C program in the following sections:

- In "Generating a C Program" on page 2638, the process of generating a C program is described.
- In "Abstract Data Types" on page 2656, implementation aspects especially concerning abstract data types, are described.
- In "Directives to the Cadvanced/Cbasic SDL to C Compiler" on page 2720, possibilities to give additional information to the Cadvanced/Cbasic SDL to C Compiler are discussed.
- The section "Using Cadvanced/Cbasic SDL to C Compiler to Generate C++" on page 2749, contains considerations on class definitions as C++ code and the utilization of the classes as C++ code in SDL tasks.
- In "Restrictions" on page 2753, the Cadvanced/Cbasic SDL to C Compiler restrictions are covered.

Generating a C Program

A generated C program can be used for several things, for example as a simulator, as an explorer, or as an application with the behavior defined by the translated SDL system. The process of generation of simulators, explorers, or applications, see below, is started in the Organizer, in the make dialog (see "Make" on page 120 in chapter 2, *The Organizer*) or by the quick buttons for simulation and validation (see "Quick Buttons" on page 180 in chapter 2, *The Organizer*).

The SDL Analyzer, which contains the C code generation facility, can also be started as a stand-alone tool. For more information about this possibility please see "The Analyzer Command Line UI" on page 2474 in chapter 54, *The SDL Analyzer*.

Process of Generating a C Program

There are four steps that must be performed to start the execution of, for example, a simulator:

1. The SDL Analyzer and its built-in Cadvanced/Cbasic SDL to C Compiler create a program expressed in C source code.
2. The generated C file (or files) is compiled.
3. The compiled file (or files) is linked together with a predefined library.
4. The executable program that is created in the link operation is started.

This process has been automated and requires no user knowledge about compiling or linking of programs. The process is initiated in the Organizer using the quick buttons for simulation and validation, or by using the *Make* dialog.

A C program can only be generated for an SDL system. The C code that constitutes the program can, however, be generated on multiple files, which means that a local change in, for example, a block diagram only requires a regeneration and recompilation of the code for that unit. The object files, (the compiled versions of the C files) for the other unchanged units can then be used in the link operation to form a new executable program. This feature in automatically used by the make facility and the quick buttons, to minimize the amount of work and thus the turn-around time, for the process from a change in the SDL system to a new simulator (for example).

The separation of the C code for an SDL system can be decided by the user. The *Edit Separation* command from the *Generate* menu is used for that purpose, see "Edit Separation" on page 137 in chapter 2, *The Organizer*. The effect on the generated file structure and some guidelines of how to use separation can be found in the section "Selecting File Structure for Generated Code – Directive #SEPARATE" on page 2721.

Executing a C Program

The generated C programs can in principle be compiled as either a simulator, an explorer, or an application. Generated applications have no further connection with the SDL Suite and are executed as any other application.

A generated simulator or explorer can however be started in two different ways:

- From the *Simulator Graphical User Interface*, which is started from the Organizer with the *SDL > Simulator UI* command and provides a graphical interface with buttons and menus and of course full connection to other SDL Suite tools. See *"Graphical User Interface" on page 2199 in chapter 49, The SDL Simulator* for more information.
- From an OS shell, just like any other executable program. The user then invokes a *command line monitor* system. If the Organizer is running when starting a simulator or explorer, the program will connect itself to the SDL Suite. If the Organizer is not running or the simulator/explorer is started with the program parameter `-nosdt`, the program will not connect itself to the SDL Suite.

The SDL Unit for Which Code is Generated

The first time a C program is generated for a system, the complete system will be selected for analysis and C code generation. After that only the unit (*system* or *block*) that is changed will be selected. Note that the lowest level of possible regeneration object is a block. That block may not be a block type, or be part of a block type or system type. The reason that a process cannot be generated without regenerating the enclosing block, is that internal process information about, for example, formal parameters are used to generate code for other processes within the same block.

- Only complete C files can be generated. If, for example, the user has specified that a *block* and a *sub-block* should be generated on the

same file, it is not possible to regenerate code only for the sub-block.

- If the file structure is changed (by, for example, changes in the *Edit Separation* command or in the #SEPARATE directives), then the complete system is regenerated.

Errors During Code Generation

Errors that may occur during code generation are internal errors. That is, errors due to not yet implemented features of SDL, and errors related to problems with opened or closed operations of files.

An error message starts with an SDT references and is followed by a description of the error, including a error number. Example:

```
ERROR 884 Not implemented: Signal refinement
```

Features

Partitioning

General Ideas

The partitioning concept is a way to divide one SDL system into several applications. As a special case this means also that it is possible to simulate and validate selected parts of a system. You should note the difference between partitioning and separation. The partitioning feature is a way to select the parts of an SDL system which should be handled, while the separation feature is a way to select the file structure for the generated files.

To select a partition (or a program) it is, in simple cases, possible to use selections in the Organizer, and in more general cases possible to work with *build scripts*, i.e. text files containing commands to the Analyzer (the syntax used when running the Analyzer stand-alone). The restriction in the Organizer view is that only one selection can be handled and that instantiation of OO types cannot be selected. In a build script on the other hand, several component commands can be used to select several parts of a system. As the component command takes an SDL qualifier as a parameter, instantiations can also easily be selected.

Using Selections in the Organizer

To start with the simple case when one block or process should be simulated, this is easy to perform directly from the Organizer: Select the proper block or process and press the *Simulate* button (or go via *Make* dialog).

> **Note:**
>
> If you already have generated a simulation from the Organizer, and want to generate a new one with other options or with another selection, you should perform a Full Make, as changes in options or selection is not handled by the build process. Otherwise compilation or link errors might be the result of the build process.

Only the system, a block or a process can be selected for simulation. Types, including procedures, are only definitions and are not executable objects, while services depend on its enclosing process and cannot be

simulated on its own because of these dependencies. Block and process instantiations can be simulated, but only using build scripts, as such objects cannot be selected in the Organizer. The discussion above is of course also valid for generation of explorers and applications.

Unconnected Diagrams

As another special case, there might be unconnected diagrams in the Organizer, i.e. objects not bound to a file. If such an object is a block, process, or a procedure, C code can anyhow be generated resulting in, for example, a Simulator or Explorer.

- If a block is unconnected, this is treated as an implicit partitioning excluding this block.
- If a process is unconnected, this is treated as an implicit partitioning excluding this process. If other processes try to “create” such a process, this will become a null-action just indicated in the textual trace. In an application, such a create action will cause a compilation error.
- If a procedure is unconnected, any call to this procedure is just indicated in the textual trace. In an application such a procedure call will cause a compilation error.

Build Scripts

In general cases, build scripts should be used to specify the build process. Using such a file there are a number of features that can be used.

- It is possible to generate code for several partitions, using independent options and potentially different code generators for different parts of the system, all in one build process.
- Each partition can consist of several objects, and objects might be instantiations.

There are two Analyzer commands, see *“Description of Analyzer Commands” on page 2476 in chapter 54, The SDL Analyzer*, that are of major interest for specifying a partition. First we have the Program command, which takes a name as parameter. Second we have the Component command, which takes a qualifier as a parameter. The Program command gives the start of a partitioning specification, while the Component command is used to select an SDL component that should be

part of the partitioning. A partition specification can of course contain a number of components. The Component command is very similar to a selection in the Organizer when running directly from the Organizer.

A program section in a build script typically starts with a Program command and ends with a Generate command.

Example 353: Build script ––––––––––––––––––––––––––––––––––

```
program MyExample
component system example/block b1
component system example/block b2/process p22
target-directory /home/jk/example/target
set-env-header on
set-modularity user
generate-advanced-c
```

In Windows, the target-directory command could, for example, be:

```
target-directory c:\example\target
```

––

The example above means that a program containing the implementation of the complete block b1 and the process p22 in block b2 is generated with the Cadvanced SDL to C Compiler. The modularity is user defined and a system header file (`.ifc` file) will also be generated.

Code from the code generators will be placed in a subdirectory with the name given in the Program command, to the directory given by target directory. If this subdirectory does not exist it will be created.

> **Note:**
>
> You should always include a target-directory command in a build script, as otherwise the target directory will depend on where the SDL Suite is started!

In the example above the generated C code can be found in the directory `/home/jk/example/target/MyExample`
(**In Windows** `c:\example\target\MyExample`).

In the example below three programs are generated for three different partitionings, also using different code generators.

Example 354: Build script with several programs ––––––––––––––––––

```
target-directory /home/jk/example/target

program MyExample
component system example/block b1
component system example/block b2/process p22
set-env-header on
set-modularity user
set-kernel SCTDEBCOM
generate-advanced-c

program MyExample1
component system example/block b2/process p21
generate-micro-c

program MyExample2
component system example/block b3
set-modularity no
generate-chipsy-chill
```

––

Analyzer commands that are of the type “set up an option” can be placed outside of the Program commands. The options actually used at the generate commands, are the options set up after executing all the commands up to the generate command. All the possibilities in the Make dialog and the Analyze dialog in the Organizer are also provided as commands in the Analyzer. Please see *“The Analyzer Command Line UI” on page 2474 in chapter 54, The SDL Analyzer*, for a list of all commands.

> **Note:**
>
> When build scripts are used, all features in the Analyzer will have its hard coded defaults, if it is not set in the build script. Preferences and your settings in the Organizer are ignored. The default values are given in *“The Analyzer Command Line UI” on page 2474 in chapter 54, The SDL Analyzer*, or you can start a stand-alone analyzer (sdtsan) directly in an OS shell and issue the commands Show-Analyze-Options and Show-Generate-Options.

See also “SDL Make” on page 121 in chapter 2, *The Organizer*, for handling of build scripts in the Organizer.

Behavior of Generated Partitioning

The basic idea is to redirect all channel going to objects not part of the current partitioning to the environment. This operation is performed by the code generator at code generation time. This means that all signals sent between objects in the partitioning and objects outside the partitioning, will be seen as signals to or from the environment. This is true everywhere, in simulations, validations, applications, in generated environment header files (.ifc files), and in generated environment functions.

Generation of Support Files

The Cadvanced/Cbasic SDL to C Compiler can generate a number of support files, together with the ordinary `.c`, `.h`, and makefiles. These files are

- System header file (`.ifc`)
- Skeleton to environment functions (`_env.c`)
- Signal number file (`.hs`)
- Coder/decoder framework files (`_cod.c`, `_cod.h`)

The generation of these files can be selected in the Organizer Make dialog, or as an Analyzer command, depending on which interface is used. The details on the system header files and the environment function can be found in "The Environment Functions" on page 2774, while the signal number file can be used to assign numbers to all signals in the system. Signal number files are most used in connection with OS integrations.

Implementation

In this section some implementation details are presented, that can be useful for understanding how a generated simulation or application behaves. Abstract data types are treated in the next section.

Time

A generated C program can be executed in two modes with respect to the treatment of time:

- *Simulated time*
- *Real time*

Simulated Time

Using simulated time, which is the most useful mode for simulations, means that the time in the simulation has no connection with the wall clock. Instead the *discrete event simulation* technique is used. This technique is based on the idea that the current value for the simulation time (Now in SDL) is equal to the time at which the currently executing event is scheduled. After one event is finished, the simulation time is increased to the time when the next event is scheduled and this event is started. Events in SDL will be process transitions, timer outputs, and signals sent to the system from the environment. As an example, the use of the discrete event simulation technique means that if the next event is a timer output scheduled one hour from now, and the next transition is allowed to execute, the timer output will occur immediately. The simulation time will be increased by one hour, but the user does not have to wait one hour.

Real Time

If real time is used, then there will be a connection between the clock in the executing program and the wall clock. In the example above the user would have to wait one hour until the timer output took place. To implement real time a clock function provided by the operating system is used. Not all systems are suitable to simulate in this way. The time scale in the system ought to be seconds or maybe minutes, not milliseconds and not hours.

At program start up the system time, SDL Now, is zero. The system clock is stopped during the time the program spends in the monitor system.

Note:

The C standard function `time` used as real time clock returns the time in seconds. It does not handle parts of seconds. The implementation of the clock can be changed by re-implementing the function SDL_Clock in `sctos.c`.

Scheduling

The process instances in the simulated system will execute transitions that consist of actions like tasks, decisions, outputs, procedure calls, etc., according to the rules of SDL. It is assumed that a transition takes no time and that a signal instance is immediately placed in the input port of the receiver when an output operation occurs.

A transition is always executed without any interrupts, if the user does not manually rearrange the ready queue using an appropriate command provided by the monitor system (Rearrange-Ready-Queue). It is possible to execute a few SDL symbols in one transition and then to re-arrange the ready queue and execute another transition. The interrupted transition can afterwards be executed to its end.

A quasi-parallel strategy for selecting transitions to be executed is thus the basic scheduling mechanism. SDL does not in itself define an execution strategy so the selected strategy is therefore an allowed, but not the only, possible strategy for execution.

As a consequence of the execution strategy, a generated simulator is not directly suited for simulation of “timing effects”, that is, situations where the time or order of actions in different process instances is of vital importance.

Example 355: Scheduling ––––––––––––––––––––––––––––––––––

An example of such a situation is: Suppose a process instance A outputs two signal instances during the same transition, one to process instance B and one to process instance C. During the corresponding transitions of B and C, a signal instance is sent to process instance D.

If the behavior of the system is dependent on the order in which the signal instances are received in the input port of D, this is a hazard situation

where the execution speed of process instances and the delay of signals in channels will determine the behavior. The way to handle such a situation would be to manually decide the order in which transitions should be executed.

As the Cadvanced SDL to C Compiler is also intended to generate applications, process priority has been introduced as an additional feature. For more information about how to assign priorities to processes see sub-section "Assigning Priorities – Directive #PRIO" on page 2739.

The Ready Queue

The ready queue is a queue containing all process instances which have received a signal that can cause a transition, but which have not yet completed that transition. The ready queue is ordered firstly according to the priority and secondly according to insert time, that is a process which will be inserted last among the processes with the same priority, but before all processes with lower priority (high priority value = low priority). A process will never be inserted before the process currently executing, as pre-emptive scheduling is not used. In more detail:

- If a process outputs a signal to another process, which immediately can receive the signal, the receiving process will be inserted into the ready queue last among the processes with the same priority, but never before the currently executing process.
- If the processes currently executing a nextstate immediately can continue to execute another transition, it will be inserted into the ready queue last among the processes with the same priority. This means that it can remain as first process in the ready queue, but it can also be re-inserted somewhere else.
- If the receiving process at a timer output immediately can execute a transition as response to the received signal, the process will be inserted into the ready queue last among the processes with the same priority. This means that it can be inserted anywhere in the queue.

Enabling Conditions and Continuous Signals

Enabling conditions and continuous signals are additional concepts in SDL. The model for these concepts use repetitive signal sending, to have the expressions recalculated repeatedly. This model is not suitable during simulation, and definitely not acceptable in an application. We

have therefore used an implementation strategy closer to the described behavior of the concepts, rather than the model used to define the concepts.

Implementation Strategy

First we distinguish between those enabling conditions and continuous signals that are dynamic and those that are static, that is containing expressions that can or cannot change their value when the corresponding process is waiting in the state. The expression in a dynamic enabling condition or continuous signal contains some part that can change its value, even though the process does not execute any statements. Or, put more precisely, it contains at least one import, view, or reference to Now.

Static enabling conditions or continuous signals do not provide any problems or any execution overhead, except that the corresponding expressions have to be calculated at nextstate operations. Dynamic enabling conditions or continuous signals, however, have to repeatedly be recalculated. The strategy selected for these expressions is to recalculate them after each transition or timer output performed by any process (and additionally also before the monitor is entered within a transition). In other words, each process waiting in a state containing a dynamic enabling condition or continuous signal executes an implicit nextstate operation between each transition or timer output performed by other processes.

Synonyms

Synonyms

An SDL synonym is implemented either as a C macro (`#define`) or as a C variable. To be translated to a macro the expression defining the value of the synonym must be:

- Of one of the predefined SDL sorts (Integer, Real etc.).
- Possible to calculate at analyze time, i.e. it may only contain literals and operators defined in the predefined SDL sorts and other synonyms which are possible to calculate at analyze time.

All other synonyms are implemented as variables given their values at program start up.

The reason for raising this question is because it is relevant to the implementation of arrays and powersets. There are two different implementations for each of these concepts, see "Array" on page 2671 and "Powerset" on page 2672. An array in SDL can either be translated to an array in C or to a linked list in C. A powerset can either be translated to a bit array in C or to a linked list. The translation method is selected by looking at the index type. If the index type is a syntype with one limited range, the array and bit array scheme is used, otherwise the linked list is used.

If a synonym translated to a variable is used in a range condition of a syntype and the syntype is used as an index sort in an array or powerset instantiation, the linked list scheme is used to implement the array or powerset. The reason for this is that the length of the array cannot depend on a variable in C.

External Synonyms

External synonyms can be used to parameterize an SDL system and thereby also a generated program. The values that should be used for the external synonyms can either be read by the generated program during start up, or included as macro definitions into the generated code. The Cadvanced/Cbasic SDL to C Compiler can handle both these cases – it is not necessary to select which way should be used for each synonym until the program is compiled.

Using a Macro Definition

To use a macro definition in C to specify the value of an external synonym, perform the following steps:

1. Write the macro definitions on a file.

   Example 356: Macro Definition ----------

   ```
   #define synonym1 value1
   #define synonym2 value2
   ```

   The synonym names are the SDL names (without any prefixes) and with any character not in letters, digits or underscore removed.

2. Introduce the following `#CODE` directive at the system level among the SDL definitions of, for example, synonyms, sorts, and signals but before any use of the synonyms.

Example 357: #CODE Directive

```
/*#CODE
#TYPE
#include "filename"
*/
```

If this structure is used, the value of an external synonym can be changed merely by changing the corresponding macro definition and recompiling the system.

> **Note:**
> When an application is created, macro definitions should be used for all external synonyms, as the function for reading synonym values stored on file is not available. (See below.)

Reading Values at Program Start up

The other way to supply the values of the external synonyms is to read the values at program start up. If there are any external synonyms that do not have a corresponding macro definition, it is possible to choose between supplying the values of the remaining external synonyms from the keyboard or to use a file containing the values.

When the application is started, the following prompt appears:

```
External synonym file :
```

- Press `<Return>` to indicate that the values should be read from the terminal.
- Or type the name of a file that contains the values and press `<Return>`.

If the user chooses to read the values from the terminal, he will be prompted for each value. In the other case the user should have created a file containing the external synonym names and their corresponding value according the following example:

Example 358: Values at Program Startup

```
synonym1 value1
synonym2 value2
```

The synonyms may be defined in any order.

Evaluation of Include Expressions

A synonym value can be followed by a simulator only comment:

```
mySynonym myValue /*SIMULATOR_ONLY*/
```

This value will not be used when evaluating include expressions.

This value will only used by the simulator when simulating the system.

Import – Export

These concepts are not implemented with the full semantics according to the model in the SDL recommendation. The model says that an imported value should be obtained using a signal interchange between the importer and exporter.

In the Cadvanced/Cbasic SDL to C Compiler we use a model where the imported value is directly obtained from the exporter, which of course makes the import operation much faster. However, the scheduling effect of the signal interchange is lost, as well as the change of SENDER in the involved processes. If these effects are important for an application, remote procedure calls can be used instead, see below.

Remote Procedure Calls

Remote procedure calls (RPC) have much in common with import/export, except that instead of obtaining one value, RPCs give the opportunity to execute a procedure in the exporting process. In the Cadvanced/Cbasic SDL to C Compiler, the model described in the SDL recommendation is used in detail to implement RPCs.

This means that a remote procedure call is translated to:

output of pCALL signal with all parameters.
nextstate in pWAIT, i.e. a implicit wait state.
input of pREPLY signal with all IN/OUT parameters.

In the exporting process there will be implicit transitions where the pCALL signal can be handled.

input pCALL.
call remote procedure with parameters from pCALL.
output pREPLY with the IN/OUT parameters.
nextstate -

For more details about this model, please see the SDL recommendation.

Procedure Calls and Operator Calls

In SDL-92 value returning procedures (and remote procedures) are introduced. This means that an SDL procedure can be called within an expression. In the Cadvanced/Cbasic SDL to C Compiler such procedure calls are implemented according to the model in the SDL recommendation, that is by inserting an extra CALL just before the statement containing the value returning procedure call. The result from the call is stored in an anonymous variable, which is then used in the expression.

Example 359: Procedure Call ----------

```
TASK i := (call p(1)) + (call Q(i,k));
```

is translated to:

```
CALL p(1, Temp1);
CALL q(i, k , Temp2);
TASK i := Temp1 + Temp2;
```

Note:

The value returning procedure calls are transformed to ordinary calls, by adding a new IN/OUT parameter for the procedure result, last in the call.

Operators which are defined using operator diagrams, are according the models in the SDL recommendation, treated exactly as value returning procedure.

External Procedures And Operators

External procedures is a extension to SDL introduced in SDL-96. An external procedure is defined in a text symbol as a procedure heading:

```
procedure test; fpar a integer; returns integer;
external;
```

Instead of giving an implementation for the procedure the keyword external is inserted. The purpose of external procedures in SDL are to specify the existence of procedures without giving their implementation.

The Cadvanced/Cbasic SDL to C Compiler will generate no code for an external procedure declaration and will translate a call to such a procedure to an ordinary C function call. It is then up to the user to provide

the C implementation of this function. Note that the code generator will in the generated function call use the name of the external procedure as it is. No prefix is inserted in this case, just as for external synonyms.

External operators are handled in the same way as external procedures. The name of the external operator is used in C just as it is. A quoted operator will cause an infix operator to be generated, while operators with ordinary names will cause C function calls to be generated.

Any

There are two different applications of any. It is possible to write

```
any(SortName)
```

within an expression, or to write just

```
any
```

in a decision. The second case, with any in a decision, is implemented in the following way:

- Simulator:

  A question in the monitor giving the user a possibility to select the path to follow.

- Explorer:

  The explorer sees this as an non-deterministic choice and selects all possible paths.

- Applications:

  Should not be used!

The first case, the any(SortName) within an expression, is implemented using a random number generator to draw a random number of the given type.

> **Note:**
>
> any(Sort) where Sort is a syntype is only implemented if the syntype contains at most one range condition which is of the form a:b, that is one limited range. If it is a syntype of a real type, e.g. Real or Time, with a range condition it is not implemented.

If any(SortName) is used for a sort violating the note above, there will be a C compilation error on the symbol ANY_SortNameWithPrefix.

This means that a user can implement any for such sorts himself by defining a C macro with this name, that implements any for the given sort. Such a macro should be inserted in the #TYPE section of a #ADT directive in the syntype.

Calculation of Receiver in Outputs

The Cadvanced/Cbasic SDL to C Compiler contains an algorithm that calculates the receiving process instance set, for outputs without TO, considering channels, signal routes, connection points, and via list. There are however a few restrictions for the algorithm:

- Outputs in process types, or in processes in block types or system types **cannot** be handled. The reason is that the same output might lead to different receivers in different instantiation.
- Paths (channels - signal routes) that lead into other units that are separate (see "Edit Separation" on page 137 in chapter 2, *The Organizer*) **cannot** be followed by the algorithm, as that would violate the separate generation scheme.
- Outputs in global procedures **cannot** be handled, as the receiver depends on the caller of the procedure.

This algorithm means that for an ordinary SDL-88 system, that is not generated using separate units, no information about the channels and signal routes are needed to direct signal to the correct receiver. For more information about the possible optimizations in applications, please see the compilation switch XOPTCHAN and the ADT PidLit ("The Data Type PIdLit" on page 3258 in chapter 62, *The ADT Library*). Please note that XOPTCHAN and PidLit is almost impossible to use if the SDL system contains system types, block types, or process types.

Abstract Data Types

This section is a reference to the abstract data types. The following topics will be discussed:

- We will have a look at the implementation of the predefined data types in SDL, see "SDL Predefined Types" on page 2658. We then discuss how user-defined abstract data types are translated, see "Translation of Sorts" on page 2665.
- Next implementation of operators and the possibility to include hand-coded C functions as implementation of the operators is presented, see "Implementation of User Defined Operators" on page 2679.
- Last, in "More about Abstract Data Types" on page 2704, we discuss more details about operators and the possibilities to include a hand-coded type definition in C to represent the SDL sort.

Removing un-used SDL Operators

When implementing an SDL system, you do not always use all available SDL operators. The Cbasic/Cadvanced SDL to C Compiler removes the declarations of unused operators, thus minimizing the code size of the generated application. Unused operators that are removed are:

- operators in predefined data types, for example substring, concatenate, length in the newtype Charstring, etc.
- operators defined in the predefined generators String, Array, Powerset, Bag
- special operators (and help functions) like assign, equal, default, make, extract, modify, free

The Cbasic/Cadvanced SDL to C Compiler performs the following steps to optimize the code:

1. Every C function that implements an operator is surrounded by an `#ifndef` definition.

Example 360 The `#ifndef` definition ––––––––––––––––––––––

```
#ifndef XNOUSE_AND_BIT_STRING
  /* function implementing the operator */
```

```
#endif
```

2. During the code generation, the usage of the operators in the translated SDL transitions is recorded.
3. The interdependencies between different operators are updated. For instance, an equal operator for a struct type may depend on equal operators for all its component types.
4. For each operator that is found to be unused, a `#define` definition is generated that removes the code for that operator. All the defines are placed in a file called *sdl_cfg.h.*

Example 361 The `#define` command

```
#define XNOUSE_AND_BIT_STRING
```

> **Note:**
> Even though the code size of the generated application is reduced, the code size of the generated C code is increased.

Manual override

In order to handle cases where operators are used invisibly from the Cbasic/Cadvanced SDL to C Compiler, for example in inline C code, you can manually override the automatic configuration of the unused operators.

In the code generation process, the Targeting Expert always generates a manual configuration file called `sct_mcf.h`. In this file you can list the unused operators that you have decided to include in the application. This is done by un-defining the previous definitions made in the `sdl_cfg.h` file.

The `sct_mcf.h` can be edited directly from Targeting Expert. Select **Edit Configuration Header File** from the **Edit** menu to open the file.

Example 362 The `#undef` command in the `sct_mcf.h` file

```
#ifdef XNOUSE_AND_BIT_STRING
#undef XNOUSE_AND_BIT_STRING
#endif
```

One section of the sct_mcf.h file, is dedicated for the manual edits. This section is marked with the text:

```
/* BEGIN User Code */
/*   END User Code */
```

The manual edits must be inserted between these to lines otherwise they will be deleted, as the sct_mcf.h file is re-generated each time you generate code. The information about the unused operators available in the sdl_cfg.h file is imported to the sct_mcf.h file. This allows you to quickly see which operators that are unused.

SDL Predefined Types

Mapping Table

Below is a table which summarizes the mapping rules between SDL and C, concerning the predefined types in SDL and their operators. Note that many of the operators are in C defined as macros, and expanded by the C preprocessor to simple operators in C.

SDL name/operator	C name/expression/operator
Boolean	SDL_Boolean
False, True	SDL_False, SDL_True
not	xNot_SDL_Boolean
and	xAnd_SDL_Boolean
or	xOr_SDL_Boolean
xor	xXor_SDL_Boolean
=>	xImpl_SDL_Boolean
=, /=	yEqF_SDL_Boolean, yNEqF_SDL_Boolean

SDL name/operator	C name/expression/operator
Character	SDL_Character
NUL SOH ...	SDL_NUL SDL_SOH ... (for all unprintable characters)
'a' 'b' ...	'a' 'b' ... (for all printable characters except ' and \)
'''', '\'	'\'', '\\'
chr	xChr_SDL_Character
num	xNum_SDL_Character
<, <=, >, >=	xLT_SDL_Character, xLE_SDL_Character, xGT_SDL_Character, xGE_SDL_Character
=, /=	xEqF_SDL_Character, xNEqF_SDL_Character
Charstring	SDL_Charstring
'aa'	SDL_CHARSTRING_LIT("Laa","aa")
mkstring	xMkString_SDL_Charstring
length	xLength_SDL_Charstring
first	xFirst_SDL_Charstring
last	xLast_SDL_Charstring
//	xConcat_SDL_Charstring
substring	xSubString_SDL_Charstring
=, /=	yEqF_SDL_Charstring, yNEqF_SDL_Charstring

SDL name/operator	C name/expression/operator
Integer	SDL_Integer
0, 1 etc.	SDL_INTEGER_LIT(0), SDL_INTEGER_LIT(1) etc.
+	xPlus_SDL_Integer
- (monodic, dyadic)	xMonMinus_SDL_Integer, xMinus_SDL_Integer
*	xMult_SDL_Integer
/	xDiv_SDL_Integer
mod	xMod_SDL_Integer
rem	xRem_SDL_Integer
float	xFloat_SDL_Integer
fix	xFix_SDL_Integer
<, <=, >, >=	xLT_SDL_Integer, xLE_SDL_Integer, xGT_SDL_Integer, xGE_SDL_Integer
=, /=	yEqF_SDL_Integer, yNEqF_SDL_Integer
Natural	SDL_Natural
Real	SDL_Real
12.45, ...	SDL_REAL_LIT(12.45, 12, 450000000)
- (monodic, dyadic)	xMonMinus_SDL_Real, xMinus_SDL_Real
+	xPlus_SDL_Real
*	xMult_SDL_Real
/	xDiv_SDL_Real
<, <=, >, >=	xLT_SDL_Real, xLE_SDL_Real, xGT_SDL_Real, xGE_SDL_Real
=, /=	yEqF_SDL_Real, yNEqF_SDL_Real

SDL name/operator	C name/expression/operator
Pid	SDL_PId
Null	SDL_NULL
=, /=	yEqF_SDL_PId, yNEqF_SDL_PId
Duration	SDL_Duration
23.45	SDL_DURATION_LIT(23.45, 23, 450000000)
+	xPlus_SDL_Duration
- (monodic)	xMonMinus_SDL_Duration
- (dyadic)	xMinus_SDL_Duration
* (Duration * Real) * (Real * Duration)	xMult_SDL_Duration, xMultRD_SDL_Duration
/	xDiv_SDL_Duration
<, <=, >, >=	xLT_SDL_Duration, xLE_SDL_Duration, xGT_SDL_Duration, xGE_SDL_Duration
=, /=	yEqF_SDL_Duration, yNEqF_SDL_Duration
Time	SDL_Time
23.45	SDL_TIME_LIT(23.45, 23, 450000000)
+ (Time + Duration) + (Duration + Time)	xPlus_SDL_Time, xPlusDT_SDL_Time
- (result: Time)	xMinusT_SDL_Time
- (result: Duration)	xMinusD_SDL_Time
<, <=, >, >=	xLT_SDL_Time, xLE_SDL_Time, xGT_SDL_Time, xGE_SDL_Time
=, /=	yEqF_SDL_Time, yNEqF_SDL_Time
IA5String	SDL_IA5String
NumericString	SDL_NumericString
VisibleString	SDL_VisibleString
PrintableString	SDL_PrintableString

SDL name/operator	C name/expression/operator
Bit	SDL_Bit
not	xNot_SDL_Bit
and	xAnd_SDL_Bit
or	xOr_SDL_Bit
xor	xXor_SDL_Bit
=>	xImpl_SDL_Bit
=, /=	yEq_SDL_Bit, yNEq_SDL_Bit
Bit_string	SDL_Bit_String
not	xNot_SDL_Bit_String
and	xAnd_SDL_Bit_String
or	xOr_SDL_Bit_String
xor	xXor_SDL_Bit_String
=>	xImpl_SDL_Bit_String
mkstring	xMkString_SDL_Bit_String
length	xLength_SDL_Bit_String
first	xFirst_SDL_Bit_String
last	xLast_SDL_Bit_String
//	xConcat_SDL_Bit_String
substring	xSubString_SDL_Bit_String
bitstr	xBitStr_SDL_Bit_String
hexstr	xHexStr_SDL_Bit_String
=, /=	yEq_SDL_Bit_String, yNEq_SDL_Bit_String

SDL name/operator	C name/expression/operator
Octet	SDL_Octet
not	xNot_SDL_Octet
and	xAnd_SDL_Octet
or	xOr_SDL_Octet
xor	xXor_SDL_Octet
=>	xImpl_SDL_Octet
shiftl	xShiftL_SDL_Octet
shiftr	xShiftR_SDL_Octet
+	xPlus_SDL_Octet
-	xMinus_SDL_Octet
*	xMult_SDL_Octet
i2o	xI2O_SDL_Octet
o2i	xO2I_SDL_Octet
/	xDiv_SDL_Octet
mod	xMod_SDL_Octet
rem	xRem_SDL_Octet
bitstr	xBitStr_SDL_Octet
hexstr	xHexStr_SDL_Octet
<, <=, >, >=	yLT_SDL_Octet, yLE_SDL_Octet, yGT_SDL_Octet, yGE_SDL_Octet
=, /=	yEq_SDL_Octet, yNEq_SDL_Octet

SDL name/operator	C name/expression/operator
Octet_string	SDL_Octet_String
mkstring	xMkString_SDL_Octet_String
length	xLength_SDL_Octet_String
first	xFirst_SDL_Octet_String
last	xLast_SDL_Octet_String
//	xConcat_SDL_Octet_String
substring	xSubString_SDL_Octet_String
bitstr	xBitStr_SDL_Octet_String
hexstr	xHexStr_SDL_Octet_String
bit_string	xBit_String_SDL_Octet_String
hex_string	xHex_String_SDL_Octet_String
=, /=	yEq_SDL_Octet_String, yNEq_SDL_Octet_String
Object_identifier	SDL_Object_Identifier
mkstring	xMkString_SDL_Object_Identifier
length	xLength_SDL_Object_Identifier
first	xFirst_SDL_Object_Identifier
last	xLast_SDL_Object_Identifier
//	xConcat_SDL_Object_Identifier
substring	xSubString_SDL_Object_Identifier
=, /=	yEq_SDL_Object_Identifier, yNEq_SDL_Object_Identifier
NULL (sort)	SDL_Null
NULL (literal)	SDL_NullValue
=, /=	yEq_SDL_Null, yNEq_SDL_Null

C Definitions

We will here discuss the types and macros supplied by the runtime library in the Cadvanced/Cbasic SDL to C Compiler for the predefined types in SDL. These macros and extern definitions for functions can be found in the file `sctpred.h`, except for the Pid sort which is handled in the file `scttypes.h`.

> **Note:**
>
> For more information about the Charstring sort, see the section "Handling of the Charstring Sort" on page 2688.

Translation of Sorts

The following data types are handled by the Cadvanced/Cbasic SDL to C Compiler:

- *Predefined Types*
- *Enumeration Type*
- *Struct*
- *Choice*
- *Array*
- *String*
- *Powerset*
- *Bag*
- *Ref, Own, Oref*
- *Syntypes*
- *Inheritance*

Predefined Types

All the predefined data types (Integer, Natural, Boolean, Character, Charstring, Real, Time, Duration, Pid, Bit, Bit_string, Octet, Octet_string, Object_identifier, IA5String, NumericString, PrintableString, and VisibleString) are completely handled. The name of these types in the generated C code will be SDL_Integer, SDL_Natural, SDL_Boolean, and so on. The translation rules for these types and their operators are discussed in more detail in the "SDL Predefined Types" on page 2658.

Enumeration Type

A sort which is not a struct and does not contain any inheritance or generator instantiation, but which contains a literal list, is seen as an enumeration type. See the example below. Such a type is translated to **int**, together with a list of defines where the literals are defined as 0, 1, 2, and so on. As in all examples in this sub-section, the prefixes, which are added to names when they are translated to C, are not shown. The prefixes are added to make sure that no name conflicts occur in the generated program. For more information about prefixes see "Names and Prefixes in Generated Code" on page 2735.

Example 363: Enumeration Type

```
NEWTYPE EnumType
  LITERALS Lit1, Lit2, Lit3;
ENDNEWTYPE;
```

is translated to:

```
typedef XENUM_TYPE EnumType;
#define Lit1 0
#define Lit2 1
#define Lit3 2
```

Where the macro `XENUM_TYPE` is defined in sctpred.c as:

```
#ifndef XENUM_TYPE
#define XENUM_TYPE int
#endif
```

This means that all enum types will be int types, except if the macro `XENUM_TYPE` is redefined by the user (to `unsigned char` for example). An enum type with 256 or more values will always be of type int and will not be affected by the macro `XENUM_TYPE`.

Struct

An SDL struct is translated to a struct in C, as can be seen in the example below.

Example 364: Struct

```
NEWTYPE Str STRUCT
  a Integer;
  b Boolean;
  c Real;
ENDNEWTYPE;
```

is translated to:

```
typedef struct {
  SDL_Integer a;
  SDL_Boolean b;
  SDL_Real c;
} Str;
```

All the properties of a struct in SDL are preserved in the C code.

The predefined operators extract! and modify! are implemented as component selections in the struct in the same way as in SDL, that is, if S is a variable of type Str, then S!a in SDL is translated to S.a in C.

The predefined operator make!, which is a constructor of a struct value, is implemented by generating a Make function in C. This means that the expression "(. 12, true, 0.22 .)" in SDL is in principle translated to the C function call Make(12, true, 0.22).

The components of a struct may be of any sort that the code generator can handle. A component may, however, not directly or indirectly refer to the struct sort itself. As an example the sort Str above may not have a component of sort Str. In such a case the translation to a C struct would not any longer be valid.

There are some extensions to SDL that are handled by the code generator. It is possible to define bit fields, i.e, to define the size of components (as in C) and to have optional components and components with initial values (as in ASN.1). Examples are shown below.

Example 365: Struct with bit fields

```
NEWTYPE str STRUCT
  a Integer : 4;
  b Integer : 4;
            : 0;
  c UnsignedInt : 2;
  d Integer;
ENDNEWTYPE;
```

is translated to:

```
typedef struct str_s {
    SDL_Integer  a : 4;
    SDL_Integer  b : 4;
    int : 0;
    UnsignedInt  c : 2;
    SDL_Integer  d;
```

```
} str;
```

Note that only Integer and UnsignedInt should be used in bit field components.

Example 366: Struct

```
NEWTYPE str STRUCT
  a, b integer;
  c    Boolean OPTIONAL;
  d    str2 OPTIONAL;
  e    Charstring := 'telelogic';
  f    arr3 := (. 11 .);
ENDNEWTYPE;
```

is translated to:

```
typedef struct str_s {
    SDL_Integer  a;
    SDL_Integer  b;
    SDL_Boolean  c;
    SDL_Boolean  cPresent;
    str2  d;
    SDL_Boolean  dPresent;
    SDL_Charstring  e;
    SDL_Boolean  ePresent;
    arr3  f;
    SDL_Boolean  fPresent;
} str;
```

Both optional components and components with initial values have a Present flag. This is according to ASN.1 and the translation of ASN.1 to SDL defined in Z.105. The present flag for a component with initial value is true if the component contains its default value otherwise false (the Present flag is used to determine code for some ASN.1 encoding scheme). The present flag for an optional component is false until the component is assigned a value. In SDL the present flags can only be accessed through operators and cannot be changed.

Union

Please see also the CHOICE concept presented below, as it usually provides a better and more secure solution to the same kind of problems.

Using the directive #UNION (see example below) it is possible to tell the Cadvanced/Cbasic SDL to C Compiler to generate a union according to the following example:

Example 367: Union ----------

```
NEWTYPE Str /*#UNION*/ STRUCT
  tag integer;
  a integer;
  b Boolean;
  c real;
ENDNEWTYPE;
```

is translated to:

```
typedef struct {
  SDL_Integer tag;
  union {
    SDL_Integer a;
    SDL_Boolean b;
    SDL_Real c;
  } U;
} Str;
```

The first component in the struct is assumed to be a tag value indicating which of the union components that are active. The tag should either be integer or an enumeration type. Tag value 0 or first enumeration literal is used to indicate that the first of the remaining components are active, and so on. On the SDL level a #UNION struct should be handled just like any other struct. It is up to the code generator to generate the correct code for operations on the struct, like assignment, test for equality, component selection, and so on.

> **Note:**
>
> It is completely up to the user to make certain that only valid components in a #UNION struct are accessed. During simulation, however, tests are inserted to ensure that only valid components are accessed.

UnionC

By using the directive #UNIONC according to the example below, it is possible to tell the Cadvanced/Cbasic SDL to C Compiler to generate a true C union.

Example 368: UnionC ----------

```
NEWTYPE Str /*#UNIONC*/ STRUCT
  a integer;
  b Boolean;
  c real;
ENDNEWTYPE;
```

is translated to:

```
typedef union {
  SDL_Integer a;
  SDL_Boolean b;
  SDL_Real c;
} Str;
```

Note:

The #UNIONC directive is not recommended for use as the Cadvanced/Cbasic SDL to C Compiler cannot give any support for checking the validity of component selection. Both the #UNION directive and the CHOICE concept discussed below are much better.

Note also that pointer types, including Charstrings are not allowed in #UNIONC structs, as it is not possible to know when to allocate and deallocate memory for such components.

Choice

Choice, which is an SDL extension originating from the needs when translating ASN.1 to SDL and which is included in SDL-2000, can be used to express a union with implicit tag.

Example 369: Choice ----------

```
NEWTYPE Str CHOICE
  a integer;
  b Boolean;
  c real;
ENDNEWTYPE;
```

is translated to:

```
typedef enum {a, b, c} StrPresent;
typedef struct {
  StrPresent Present;
  union {
    SDL_Integer a;
    SDL_Boolean b;
    SDL_Real c;
  } U;
} Str;
```

The component Present, which is the tag field, and its type (`StrPresent` in the example above) are both available in SDL. The Present component can in SDL be accessed, but not changed, through:

- component selection, i.e. by `Variable!Present`, i.e. it is possible to for example test: `V!Present = a`
- the operators aPresent, bPresent, or cPresent, which returns true or false depending on if the component is active or not.

The Present component is automatically set by the code generator when a component in the choice is given a value.

Note that during simulations and validations, it is automatically tested that a component “is present” when an attempt is made to access the component. A run-time error is issued if this is not the case.

Array

Instantiations of the predefined generator array can be handled by the code generator with the following restriction: The component and index sort may be any sorts that the code generator can handle, but may not directly or indirectly refer to the array type itself (see also the previous paragraph on struct).

If the index sort is a discrete sort, with one closed interval of values, that is of the following sorts:

- Character
- Boolean
- Octet
- Bit
- A sort that is considered as an enumeration type
- Syntypes of integer, character, Boolean, Octet, Bit, and enumeration types. The subtypes may only have one range condition that specifies a closed interval of values,

then the SDL array is translated to a struct containing an element which is an array in C.

If the index sort is **not** one of the sort in the enumeration above, the SDL array is translated to a linked list. The list head contains the default value for all possible indexes, while the list elements contain value pairs, (index_value, component_value), for each index having a component value not equal to the default value.

Example 370: Array ––

```
SYNTYPE Syn = integer
  CONSTANTS 0:10
```

```
ENDSYNTYPE;

NEWTYPE Arr ARRAY(Syn, real)
ENDNEWTYPE;
```

is translated to:

```
typedef SDL_Integer Syn;
typedef struct {
 SDL_Real A[11];
} Arr;
```

All the properties of an array in SDL are preserved in the C code.

The predefined operators extract! and modify! are implemented as component selection of the array in C in the same way as in SDL, so if AVar is a variable of type Arr, and Index is a valid index expression, then `AVar(Index)` in SDL is translated to `AVar.A[Index]` in C. In the case of a link list implementation of the array, component selection is made through function calls.

The predefined operator make!, which is a constructor of an array value, is implemented by generic Make function in C.

String

Instantiations of the predefined generator string can be handled by the code generator with the following restriction: The component sort may be any sorts that the code generator can handle, but may not directly or indirectly refer to the string type itself.

There are two translation schemes for Strings. The directive #STRING decide whether the string should be translated to linked list or to an array. For the #STRING directive please see "Alternative Implementations of the String Generator – Directive #STRING" on page 2746.

Strings are translated to linked list containing one element for each element in the string value. Operations and component selection in string sorts are fully supported.

Powerset

Instantiations of the predefined generator powerset can be handled by the code generator with the following restriction: The component sort may be any sorts that the code generator can handle, but may not directly or indirectly refer to the powerset type itself.

There are two translation schemes for powersets. If the component sort fulfills the conditions for index sorts mentioned in the subsection about arrays above ("Array" on page 2671), an array of 32-bit integers are used. Each bit will be used to represent a certain element whether it is a member of the powerset or not. If this is not the case, a linked list of all elements that are member of the set, is used to represent the powerset. All the available operations defined for Powersets in SDL are supported.

Bag

The Bag generator, which is introduced in SDL in Z.105, i.e. in the mapping from ASN.1 to SDL, is similar to powerset. However, it is possible to have several elements with the same value in a bag. A bag is always translated into a linked list, with one element for each value that is a member of the bag. Each element contains the value and the number of occurrences of this value.

Ref, Own, Oref

These generators represent pointers with different properties. They are all translated to pointers in C.

Syntypes

Syntypes may be defined for any sort that the code generator can handle, giving a new name for the sort and possibly a new default value for variables of the sort. Range conditions that restrict the allowed range of values are also allowed.

A syntype is translated to a type equal to the parent type using typedef. The check that a variable of a syntype is only assigned legal values is implemented in a test function that is generated together with the type definition. An attempt to assign an illegal value to such a variable will be reported as an SDL dynamic error. If the syntype is can be used as index sort in an array and the generated type in C would become an array, there will also be a test function that can be used to check that an index value is within its range in an array component selection.

Example 371: Syntypes ––––––––––––––––––––––––––––––––––––

```
SYNTYPE Syn = integer
  CONSTANTS 0:10
ENDSYNTYPE;

SYNTYPE Syn2 = integer
```

```
  CONSTANTS <0, =2, >=10
ENDSYNTYPE;

SYNTYPE Arr1 = Arr
  DEFAULT (. 2.0 .);
ENDSYNTYPE;
/* Arr defined above */
```

is translated to:

```
typedef SDL_Integer Syn;
typedef SDL_Integer Syn2;
typedef Arr Arr1;
```

Inheritance

A type that inherits another type is translated to a type equal to the parent type using a typedef.

Default Values

Default values are fully supported for all sorts that the code generator can handle, both if a default value is given in a sort definition and if an initial value is given in a variable definition (DCL).

Default values will also be assigned to all variables and components which do not have a default value specified in SDL. The reason for this is to avoid handling undefined variables in C, which might give serious problems and unexpected behavior of an executing program. The values selected by the code generator in such a case can be found below.

Note:

This is a deviation from SDL-92. It means that the generated program does not handle the value undefined for any type.

If no default value is given in the sort and no start value is given in the data definition (DCL) for a variable, the variable will be set to 0 by using a memset to 0.

Operators

In SDL-92, it is possible to define the behavior of operators in ADTs directly in SDL, using operator diagrams or operator implementations in SDL textual form. Such operators are translated to C by the Cadvanced/Cbasic SDL to C Compiler, and none of what is said below is

valid for such an operator. It is also possible to specify that an operator is external. In this case the code generator assumes that a C function with the name used in SDL exists and translates calls to the external operator directly to calls to the C function.

A user defined operator in an SDL sort definition, which is not defined by an operator diagram, is translated to a C function which asks the user for the result of the operation. At a call of an operator, the user is supplied with information describing what the operator and the sort are called, and given information about the parameter values. You are then requested to answer with the result value. If you press <Return> at the prompt for the result, the default value of the actual result type is returned. If the operator does not have a result type, no question is asked.

Example 372: Operator

```
Operator Op in sort S is called.
Parameter 1: true
Parameter 2: 10
Enter value (integer) : 12
```

assuming that newtype S contains an operator

```
Op: Boolean, Integer -> Integer;
```

More about operator implementation, both parameter passing and how to include implementations written in C can be found in the next two sections.

Literals

In sorts that are translated to enumeration types in C, literals are obviously handled by the code generator. In sorts that are not enumeration types, literals are treated as operators without parameters and are handled in exactly the same way as user defined operators.

> **Note:**
> The Cadvanced/Cbasic SDL to C Compiler does not permit naming of literals using name class literals or character strings.

Axioms and Literal Mappings

Axioms and literal mapping are allowed by the code generator in sorts, but are completely ignored.

Parameter Passing to Operators

For performance reasons the data types in SDL have been divide in two groups, simple, small types that are passed as values and structured, larger types that are passed as references (addresses).

Types passed as addresses (structured types)
Bit_string
Octet_string
Object_identifier
Struct types (including #UNION, #UNIONC)
Choice types
Instantiations of generator Powerset
Instantiations of generator Bag
Instantiations of generator Array
Instantiations of generator String
Instantiations of generator Carray
Syntypes of a type in this list
Types that inherit a type in this list

Table 1: Types Passed as Addresses

Types passed as values (simple types)
Integer
Real
Natural
Boolean
Character
Time
Duration
PId
Charstring
Bit
Octet
IA5String
NumericString
PrintableString
VisibleString
NULL
Enumeration types
Instantiations of generator Ref, Own, ORef
Syntypes of a type in this list
Types that inherit a type in this list

Table 2: Types Passed as Values

Note:

For types represented as pointers (Charstring, including its syntypes, Ref, Own, ORef), the pointers, not the addresses of the pointers, are passed as parameters.

The parameter passing for operators implemented in C works as follows (for Cmicro the mechanism described below is also used for operator diagrams and procedures):

In parameters:

- Passed as a value in C if the type is in the list “Passed as value”. This means that the parameter type in C is the same type as in SDL.
- Passed as an address in C if the type is in the list “Passed as address”. This means that the C parameter is (SDL_type *) if the type in SDL is SDL_type.

In/Out parameters:

Parameters are always passed as addresses, i.e the C parameter is (SDL_type *) if the type in SDL is SDL_type.

Operator result:

- If the result type is in the list “Passed as value”, the C function result type will be the same as in SDL.
- If the result type is in the list “Passed as address”, two things are changed. Firstly, the C result type will be (SDL_type *), i.e the result will be an address. Secondly, an extra parameter is inserted last in the C function. This parameter is also of type (SDL_type *) and is used as a location to store the result of the function. At an operator call, a “dummy” variable should be passed as the actual parameter. The C function can then use this to store the result of the operator and should return the variable again as result.

Example 373: ––

Assume that struct1 is a newtype struct in SDL.

```
operators
 X : integer, in/out integer -> integer;
 Y : struct1, in/out struct1 -> struct1;
```

The C prototypes for these operators are:

```
SDL_Integer X (SDL_Integer, SDL_Integer *);
struct1 * Y (struct1 *, struct1 *, struct1 *);
```

The example implementations are:

```
SDL_Integer X
  (SDL_Integer Param1, SDL_Integer *Param2)
```

```
{
  *Param2 = *Param2+Param1;
  return *Param2;
}

struct1 * Y (struct1 *Param1,
             struct1 *Param2,
             struct1 *Result)
{
  /* implementation assuming struct1 to contain
     two integers */
  (*Param2).comp1 = (*Param2).comp1+(*Param1).comp1;
  (*Param2).comp2 = (*Param2).comp2+(*Param1).comp2;
  *Result = *Param2;
  return Result;
    /* always return the last, extra, parameter */
}
```

Note: VERY IMPORTANT

As IN parameters are passed as addresses for structured types, changing such a parameter inside the operator might have undesired effects. A variable passed as actual parameter is then also changed. If you want to change the formal parameter copy it first to a operator local variable.

For Cadvanced/Cbasic this rule applies to operators implemented in C. For Cmicro this rule also applies to operators and procedures defined in SDL.

Implementation of User Defined Operators

Including Implementations of Operators

In a previous subsection, the default behavior of the Cadvanced/Cbasic SDL to C Compiler concerning operators (not defined in operator diagrams) and literals were described. If you do not specify otherwise interactive functions are generated, which, in each case, will ask you for the operator result or literal value. This is a fast way of getting started, but you will probably find it tedious in the long run, especially if you are using abstract data types extensively. To cope with this problem and to make it possible to generate applications, the code generator offers a possibility to include implementations written in C of the operator and

literal functions. This possibility can be used as an alternative to operator diagrams or operators defined using SDL textual form, where the operator is defined directly in SDL.

When the choice between an implementation in SDL or in C is to be made there are a few things to consider:

- There is always problems when mixing languages, for example how are C names for SDL entities constructed.
- Checking of SDL is performed by the SDL Analyzer, which will find problems much earlier than the C compiler checking C code. Also pointing to the error will be more accurate in SDL.
- SDL implementations will be more portable and might benefit from future improvements in the SDL Compiler.
- The risk for backward compatibility problems in future releases of the SDL Suite will be less for an SDL implementation.
- However, a C implementation might be more efficient or you might already have a corresponding C function.

So it is not obvious if SDL or C implementations should be used. However, we recommend SDL if there are no specific reasons for using C. Note also that the SDL extension described in "Grammar for the Algorithmic Extensions" on page 146 in chapter 3, *Using SDL Extensions, in the SDL Suite 6.2 Methodology Guidelines* could be very useful when writing implementations in SDL.

It is possible to choose between two alternatives to implement operators and literal functions:

- Q (question)
  This is the default value and specifies that the code generator should generate the interactive routines describe above.
- B (body)
  This specifies that the code generator should generate the heading of the operator and literal functions, while the user must supply the bodies of the functions.

Note:

The C functions are divided into a function heading (extern or static declaration) and a function body.

An example of a function heading (extern declaration) is:

Example 374: Implementing an Operator

```
extern SDL_Integer Max
  (SDL_Integer Para1,
   SDL_Integer Para2);
```

while the corresponding function body is:

```
SDL_Integer Max
  (SDL_Integer Para1,
   SDL_Integer Para2)
{
 if (Para1 > Para2)
   return Para1;
 return Para2;
}
```

The main reason for this division of functions into heading and body is the separate compilation scheme used in C. If, for example, an abstract data type is defined in a system and used in a process in the system, and the process is generated on a separate file, then there has to be a module interface file (a `.h` file) for the system containing the external interface (types, extern declarations of functions and so on). The interface file should then be included in the file generated for the process.

Even if separate compilation is not used, the division of functions into heading and body is useful. By having static declarations of the functions, the order in which functions must be defined is relaxed. If static declarations were not used, a function could only call the functions that are defined textually before the actual function.

To select the way the Cadvanced/Cbasic SDL to C Compiler should generate code for operators and literals, code generator directives are used. A code generator directive is an SDL comment with the first characters equal to '#', followed by a sequence of letters identifying the directive. In this case the letters are ADT (for Abstract Data Type) and OP (for operator). An ADT directive and a OP directive should thus look like:

```
/*#ADT */ /*#OP */
```

The text is not case sensitive.

OP directives are recognized at two different positions in an abstract data type:

- Directly after the name of a literal
- Directly after the semicolon ending the definition of an operator.

ADT directives are recognized immediately before the reserved word ENDNEWTYPE (or ENDSYNTYPE).

Example 375: Implementing an Operator (#ADT) ––––––––––––––––

```
NEWTYPE Str STRUCT
  a integer;
  b Boolean;
  c real;
ADDING
LITERALS
  Lit1 /*#OP */,
  Lit2 /*#OP */;
OPERATORS
  Op1 :Str,integer -> Str; /*#OP */
  Op2 :Str,Boolean -> Str; /*#OP */
/*#ADT */
ENDNEWTYPE;
```

––

At each of the positions after a literal name or operator definition, there is a possibility to specify how this literal or operator should be implemented. In the directive immediately before ENDNEWTYPE the default implementation technique can be given. When the code generator determines how to generate code for a literal or an operator, it first looks for an OP directive after the literal name or operator definition. If no such directive is found it looks for a directive immediately before ENDNEWTYPE. If no ADT directive is found here, the generation technique Q (question) is assumed. For Cmicro, Q is not used, so the default is B.

An OP or ADT directive specifying a generation technique should have the following structure:

```
/*#OP (B) */ /*#ADT (B) */
```

The letter between the parentheses should be either Q (question) or B (body). The interpretation of Q and B was explained earlier. If B has been specified for any operators or literals, then the C code for these functions must be supplied by the user. This code should be placed in the #BODY section in the ADT directive, according to the following example:

Example 376: Implementing an Operator (#ADT) ––––––––––––––––

```
/*#ADT (B)
#BODY
C code, representing bodies of functions
```

```
*/
```

The section name, i.e #BODY, must be given on a line of its own and must have the # character in the first position of the line. Upper case and lower case letters are as usual considered to be equal. If the section is empty, the section name can also be removed.

Note:

The Cadvanced/Cbasic SDL to C Compiler will **not check the consistency** between the specification of implementation techniques and the actual code included in the body section. This check is, together with checking the C code for syntactic and semantic errors, **left to the C compiler**.

Unfortunately it is not possible to have C comments within the code that is included in a #ADT directive, as SDL and C use the same symbols for start and end of comments. If a C comment is included, the SDL Analyzer will consider the end of the C comment as the end of the SDL comment. Instead a C macro called COMMENT can be used according to the examples below. Note that there might be some compiler dependent restriction of the character set allowed within the COMMENT macro. For example, the character ';' might not be allowed.

Example 377: Comment in ADT

```
COMMENT(This is a comment)
COMMENT(These comments may not contain commas \
        and should have a backslash at each \
        line break)
COMMENT((By having double parenthesis, any text
         can be entered into the comments. Some
         compilers might not allow everything.))
```

The function headings representing literals and operators are determined by their corresponding definition in SDL. The number of parameters, their types, the result type of the function and function name are all defined in SDL. In the example above, where the struct Str is defined, there are two literals (Lit1 and Lit2) and two operators (Op1: Str, integer –> Str; and Op2: Str, Boolean –> Str;). The type Str will be passed as an address, so the parameter passing rules described previously have to be applied. The function heading of the corresponding C functions should be:

Example 378: Implementing an Operator ––––––––––––––––––––––

```
extern Str* Lit1 (Str*);
extern Str* Lit2 (Str*);
extern Str* Op1 (Str*, SDL_Integer, Str*);
extern Str* Op2 (Str*, SDL_Boolean, Str*);
```

––

The function bodies, which should be supplied by the user if B is specified in the OP or ADT directive, are ordinary C functions.

Example 379: Implementing an Operator ––––––––––––––––––––––

```
Str* Lit1 (Str* Result)
{
 Result->a = 2;
 Result->b = false;
 Result->c = 10.0;
 return Result;
}

Str* Op1 (Str* P1, SDL_Integer P2, Str* Result),
{
  *Result = *P1;
  Result->a = P1->a + P2;
  return Result;
}
```

––

Before it is possible to give a complete example of an abstract data type with implementation of its operators supplied as C functions, it is necessary to look at the problem of names. When a name of some object in SDL is translated to C, a suitable sequence of characters, a prefix, is added to the SDL name, to make the name unique in the C program, see also "Names and Prefixes in Generated Code" on page 2735. This strategy is selected in the Cadvanced/Cbasic SDL to C Compiler to avoid name conflicts in the generated code, but it makes it also impossible to predict the full name of, for example, a type or a function, in the generated program. To handle this problem the user can tell the code generator to translate a name in the C code in the same way as SDL names are otherwise translated. This is specified by enclosing the SDL name between '#(' and ')' in the C code. The two functions in the previous example and their headings would then become:

Example 380: Including SDL name in C Code ––––––––––––––––––

```
extern #(Str)* #(Lit1) (#(Str)*);
extern #(Str)* #(Op1) (#(Str)*, SDL_Integer,
                       #(Str)*);
```

```
#(Str)* #(Lit1) (#(Str)* Result)
{
 Result->a = 2;
 Result->b = false;
 Result->c = 10.0;
 return Result;
}

#(Str)* Op1 (#(Str)* P1, SDL_Integer P2,
             #(Str)* Result),
{
  *Result = *P1;
  Result->a = P1->a + P2;
  return Result;
}
```

This facility to access an SDL name in C code is described in more detail in the section "Accessing SDL Names in C Code – Directive #SDL" on page 2725. A few observations concerning the example above might be appropriate:

1. The predefined sorts in SDL, that is for example integer, natural, Boolean have the names SDL_Integer, SDL_Natural, SDL_Boolean, and so on in the generated code. These types should not be enclosed between '#(' and ')'.

2. The component names of a struct are unchanged in the struct implementation in C, which means that struct components should not be enclosed between '#(' and ')' either.

Two Examples of ADTs

We now give two complete examples of abstract data types.

Example 381: ADT Example----------

```
NEWTYPE Str STRUCT
  a Integer;
  b Boolean;
  c Real;
  ADDING LITERALS
    Lit1;
  OPERATORS
    Op1 : Str, Integer -> Str;
    Op2 : Str, Boolean -> Str;
/*#ADT (B)
#BODY
#(Str)* #(Lit1) (#(Str)* Result)
```

```
{
  Result->a = 2;
  Result->b = SDL_False;
  Result->c = 10.0;
  return Result;
}

#(Str)* #(Op1) (#(Str)* P1, SDL_Integer P2,
                #(Str)* Result)
{
  *Result = *P1;
  Result->a = P1->a + P2;
  return Result;
}

#(Str)* #(Op2) (#(Str)* P1, SDL_Boolean P2,
                #(Str)* Result)
{
  if (P2)
    *Result = *P1;
  else
    (void)#(Lit1)(Result);
  return Result;
}
*/
ENDNEWTYPE;
```

The example above should be compared with the same example written in SDL. Note that the literal in the previous example is replaced with an operator without parameters. The algorithmic extensions described in "Grammar for the Algorithmic Extensions" on page 146 in chapter 3, *Using SDL Extensions, in the SDL Suite 6.2 Methodology Guidelines* is also used as they provide a powerful way to write textual algorithms.

Example 382: ADT Example in pure SDL

```
NEWTYPE Str STRUCT
  a Integer;
  b Boolean;
  c Real;
  OPERATORS
    Lit1 : -> Str;
    Op1 : Str, Integer -> Str;
    Op2 : Str, Boolean -> Str;
OPERATOR Lit1 RETURNS Str
{
   RETURN (. 2, false, 10.0 .);
}
OPERATOR Op1 FPAR P1 Str, P2 Integer RETURNS Str
{
   DCL Result Str;
   Result := P1;
```

```
    Result!a := P1!a + P2;
    RETURN Result;
}
OPERATOR Op2 FPAR P1 Str, P2 Boolean RETURNS Str
{
  IF (P2)
    RETURN P1;
  RETURN Lit1;
}
ENDNEWTYPE;
```

Example 383: ADT Example

```
SYNTYPE Index = Integer CONSTANTS 1:10
ENDSYNTYPE,

NEWTYPE A Array(Index, Integer)
  ADDING LITERALS
    Zero /*#OP (B) */;
  OPERATORS
    Add : A, A -> A; /*#OP (B) */
    Sum : A -> Integer;
/*#ADT()
#BODY
#(A)* #(Zero) (#(A)* Result)
{
  SDL_Integer i = 0;
  GenericMakeArray(Result,
    (tSDLTypeInfo *)&ySDL_#(A), &i);
  return Result;
}

#(A)* #(Add) (#(A)* P1, #(A)* P2, #(A)* Result)
{
  int I;
  for (I = 1; I<=10; I++)
    Result->A[I] = P1->A[I] + P2->A[I];
  return Result;
}
*/
ENDNEWTYPE;
```

Note that no body is supplied for the operator Sum as the default implementation strategy for operators, which should be used for Sum, is Q (question). The `GenericMakeArray` function used to implement the literal is a generic function that constructs array values. The details for this function will be described later in this section.

For more information about the functions and types (supplied by the runtime library in the Cadvanced/Cbasic SDL to C Compiler and con-

tained in generated code) that can be useful when implementing operators in C, see "SDL Predefined Types" on page 2658, and last in "More about Abstract Data Types" on page 2704.

Error Situations in Operators

In the C function used to implement operators (and literals) it is possible to define error situations and handle them as ordinary SDL run-time errors. The C library function xSDLOpError, with the following prototype:

```
extern void xSDLOpError(
  char *OpName,
  char *ErrText )
```

can be used for this purpose.

Example 384: Error Handler in Operator

Example of use:

```
  if ( strlen(C) <= 1 ) {
#ifdef XECSOP
    xSDLOpError("First in sort Charstring",
                "Charstring length is zero." );
#endif
    return SDL_NUL;
  } else
    return C[1];
```

This is a simplified version of the test in the function for the operator First in the sort Charstring. Here the error situation is when we try to access the first character in a charstring of length 0. In this case the xSDLOpError is called and a default value is returned (NUL). By including the xSDLOpError call between `#ifdef XECSOP - #endif` the function is only called to report the error if error checks are turned on. The first parameter to xSDLOpError should identify the operator and the sort, while the second parameter should describe the error.

Handling of the Charstring Sort

The SDL sort Charstring is implemented as char * in C.

Note:

This means that the value NUL (ASCII character 0) **cannot** be part of a Charstring, as this value is used as string terminator in C (this is checked by the library functions for Charstring).

The code generator and the library functions for the Charstring operators use the first character (index 0) in the C string to indicate the status of the string. If the first character is:

- 'V'
  the string is assigned to an SDL variable and may not be changed in any way.
- 'L'
  the string is a C char * literal, and may of course not be changed.
- 'T'
  the string is a temporary result from a function returning a Charstring. This memory should either be assigned to an SDL variable or returned to the pool of free memory.

All the library functions for Charstrings handle memory in an appropriate way. A user only has to take the extra character in to account, when Charstrings are handled in C. Any Charstring function parameters having a 'T' as first character must be handled according to the discussion above. A function that returns a Charstring and that creates new temporary memory to store the result, should assign the value 'T' to the first character in the Charstring.

As pointers and dynamic memory are used to implement Charstrings, it is necessary to be careful when Charstrings are handled in C code, which we show in two examples.

Example 385: Equal Test on Charstring Sort

If the `C` operator `==` is used to check if two charstrings are equal, then the actual test that is performed is to see if the two pointer values to the data areas representing the characters in the string are equal.

To check if the characters in the charstrings are equal the equal function should be used:

```
yEqF_SDL_Charstring
```

Example 386: Assignment on Charstring Sort

If the C assignment operator, =, is used to assign the value of one charstring variable (C1) to another charstring variable (C2), then two things will go wrong:

1. The memory used to represent the old value of C1 is lost and can never be reused.
2. C1 and C2 now refer to the same memory area, which means that if one of the variables is changed the other will also be changed. This leads to unpredictable behavior of the program.

The correct way to handle assignment of charstrings is to use the routine:

```
yAssF_SDL_Charstring
```

The problems mentioned above can of course also occur if a struct or array containing charstring components (or subcomponents) is handled carelessly. It is, for example, necessary to use the generic equal and assign functions to perform equal test and assignment.

To avoid problems one should be aware that Charstring is implemented as char * in C and take the consequences thereof. There are a number of help functions (that implement the operators for the Charstring sort) supplied in the runtime library that might be helpful when handling Charstrings. See "SDL Predefined Types" on page 2658).

Other Types Containing Pointers

The principal discussion about Charstrings in the previous section is also relevant for all other types containing pointers. Such types are:

- Bit_string
- Octet_string
- Object_identifier
- Strings (not #STRING)
- General Arrays
- General Powersets
- Bags

All these types contain a boolean component, IsAssigned, that gives the status of the data area. IsAssigned serves the same purpose as the first extra character in a Charstring and has to be treated in a similar way.

- IsAssigned equal to false means that this data area is a temporary result from a function returning the data type. This memory should either be assigned to an SDL variable or returned to the pool of free memory.
- IsAssigned equal to true means that this value is assigned to a variable and may not be changed in any way. It can also mean that the value is part of (i.e. is assigned to) a larger data structure.

External Properties

As an alternative to the #ADT directive, which is a comment, the external properties clause in a newtype can be used as container for this information. See the following example:

Example 387: External Properties in a Newtype ––––––––––––––––––

```
NEWTYPE Str STRUCT
  a integer;
  b Boolean;
  c real;
  ADDING LITERALS
    Lit;
  OPERATORS
    Op1 : Str, integer -> Str;
    Op2 : Str, Boolean -> Str;

ALTERNATIVE C;
#ADT (B)
#BODY
 some appropriate C code
ENDALTERNATIVE;

ENDNEWTYPE;
```

––

The #ADT directive, without the /* */ can be placed between ALTERNATIVE C; and ENDALTERNATIVE.

Note:

According to the syntax of SDL, if you have an external properties clause (i.e. alternative - endalternative), you **cannot**, in the same newtype, **have operator diagrams**, axioms, or literal mappings.

More about Operators

For an operator in an abstract data type, not only B (body) or Q (question) may be specified. The following choices are available:

- Q (question)
  This is the default value and specifies that the code generator should generate the interactive routines describe above.
- B (body)
  This specifies that the code generator should generate the heading of the operator or literal function, while the user should supply the body of the function.
- H (heading)
  This specifies that the code generator should neither generate the heading nor the body of the operator or literal function. The user is assumed to supply the necessary code.
- S (standard)
  This is used to indicate that a standard function or operator is available in the target language, which should be used as implementation of the SDL operator (literal). No function heading or function body is generated. In expressions where such an operator is used, no prefix is added to the SDL name during the translation, but the SDL name is used as it is (if no #NAME directive is present).
- P (prefix) or
  I (infix)
  where P is the default value. These letters are used to indicate if the operator should be used as a function or an operator:
  - As an operator: `a+1  a==4  -a`
  - Or as a function call: `sin(a) power(a, 3)`

Note:

As C does not include the possibility to have user defined operators, I (infix) is only adequate together with S (standard).

For each operator one of the letters B, Q, H, S and one of the letters P, I should be supplied, either in a #OP directive, or in a #ADT directive, or as the defaults Q and P; for literals P and I have no meaning.

The purpose of S is straight forward and easy to understand, but H might require some explanation. H means that the code generator will not gen-

erate any code for the operator, which leaves the user with a number of possibilities:

- By not including any code for an operator, the user may skip the code for an unused operator.
- There might already exist external declarations for a number of operators in a `.h` file that should be used instead of the generated headings.

Example 388: Using S (Standard Function or Operator) ––––––––––

Example of usage of S (standard)

```
"+" : integer, real -> real; /*#OP (SI) */
sin : real -> real; /*#OP (SP) */
```

An SDL expression using these operators:

`sin(a + 7.0)` will be translated to: `sin(zh723_a + 7.0)`

––

These examples show how standard functions in the target language can be directly utilized in abstract data types. In C, it is often easiest to use #OP(HP) for such special cases, and implement the operator in the #HEADING section as a C macro transforming the call to the appropriate syntax.

Generic Functions

Type Info Nodes

A generic function can perform a certain task for several different types. To be able to write generic functions, type-specific information for the types must be made available. This type of information could be, for instance, size of the type, component types for structured types and component offsets. This information is provided by the *type info nodes*.

A type info node is a struct that contains information that defines the type. Each type has a corresponding type info node. Each type info node contains two sections. The first section contains a sequence of general components that is identical for all type info nodes. The second section is an individual type-specific sequence of components that defines each unique type.

Every newtype or syntype introduced in SDL will be described by a type info node in the generated C code. For the predefined data types the following type info nodes can be found in `sctpred.h` and `sctpred.c`:

```
extern tSDLTypeInfo ySDL_SDL_Integer;
extern tSDLTypeInfo ySDL_SDL_Real;
extern tSDLTypeInfo ySDL_SDL_Natural;
extern tSDLTypeInfo ySDL_SDL_Boolean;
extern tSDLTypeInfo ySDL_SDL_Character;
extern tSDLTypeInfo ySDL_SDL_Time;
extern tSDLTypeInfo ySDL_SDL_Duration;
extern tSDLTypeInfo ySDL_SDL_PId;
extern tSDLTypeInfo ySDL_SDL_Charstring;
extern tSDLTypeInfo ySDL_SDL_Bit;
extern tSDLTypeInfo ySDL_SDL_Bit_String;
extern tSDLTypeInfo ySDL_SDL_Octet;
extern tSDLTypeInfo ySDL_SDL_Octet_String;
extern tSDLTypeInfo ySDL_SDL_IA5String;
extern tSDLTypeInfo ySDL_SDL_NumericString;
extern tSDLTypeInfo ySDL_SDL_PrintableString;
extern tSDLTypeInfo ySDL_SDL_VisibleString;
extern tSDLTypeInfo ySDL_SDL_Null;
extern tSDLGenListInfo ySDL_SDL_Object_Identifier;
```

For a user-defined type the type info node will have the name

```
ySDL_#(TypeName)
```

Generic Assignment Functions

Each type in SDL has access to an assignment macro `yAssF_typename`. Examples for type Boolean and for a user-defined type A:

```
#define yAssF_SDL_Boolean(V,E,A)  (V = E)

#define yAssF_A(V,E,A)  yAss_A(&(V),E,A)
#define yAss_A(Addr,Expr,AssName) \
   (void)GenericAssignSort(Addr,Expr,AssName,
                           (tSDLTypeInfo *)&ySDL_A)
```

This macro is used in the generated code (and in the kernel) at each location where an assignment should take place. The three macro parameters are:

- **V**: the variable on the left hand side
- **E**: the expression on the right hand side
- **A**: an integer giving the properties of the assignment

This macro will either become an assignment statement in C or a call of an assignment function. An assignment statement will be used if assignment is allowed according to C for the current type and if it has the correct semantics comparing with assignment in SDL.

If assignment is not possible to use, the assign macro will become a call to an assignment function. The basic generic assignment function can be found in `sctpred.c` and `sctpred.h`:

```
extern void * GenericAssignSort(void *, void *,
   int, tSDLTypeInfo *);
```

where:

- The first parameter is the address of the variable on the left hand side.
- The second parameter is the address of the expression on the right hand side.
- The third parameter is the properties of the assignment
- The fourth parameter is the type info node for the actual type.

GenericAssignSort returns the address passed as the first parameter.

The GenericAssignSort function performs three tasks:

1. The old value on the left hand side variable is released, if that is specified in properties of the assignment and if the value contains any pointers.
2. The value is copied from the expression to the variable. If possible this is performed by the function memcpy, otherwise special code depending on the kind of type is executed.
3. The IsAssigned flags are set up for the variable according to the properties of the assignment.

Special treatment of Charstring and instantiations of the Own generator has made it necessary to introduce specific wrapper functions that in their turn call GenericAssignSort for these types:

```
extern void xAss_SDL_Charstring (SDL_Charstring *,
   SDL_Charstring, int);
extern void * GenOwn_Assign (void *, void *, int,
   tSDLTypeInfo *);
```

An GenericAssignSort function must consider the following questions in order to handle the objects correctly.

How should one copy the object?

This is very important because performing the wrong action will lead to memory leaks or access errors. Three different possibilities exist:

- **AC**: always copy the referenced object.
- **AR**: always copy the pointer, i.e reusing the referenced object.
- **MR**: copy pointer if the object is temporary **or** copy object if not temporary.

What should be the status of the new object?

This is a preparation for the next operation on this object so the correct decision can be made according to the first question. Two different possibilities exists:

- **ASS**: an object should become assigned if it is assigned to a variable and needs to be copied in future assignments, i.e corresponds to the values ‘V’ and ‘L’ for the first character in a C- string representing the Charstring sort. A typical case is a normal assignment statement in SDL.
- **TMP**: an object should become temporary if it is not assigned to any persistent variable and therefore should not be copied in subsequent assignments, i.e corresponds to the value ‘T’ for the first character in a C-string representing the Charstring sort. A typical case is a result value from an operator.

What should be done with the old value referenced by the left hand side variable?

Normally free should be performed on the value, as otherwise there would be a memory leak. However, when initializing a variable, no free ought to be performed, as free might be called on a random address. Two different possibilities exists:

  - **FR**: free old value.
  - **NF**: do not free old value.

The third assignment property parameter in the GenericAssignSort function should be given a value according to the ideas given above, preferably using the macros indicated.

```
#define XASS_AC_ASS_FR  (int)25
#define XASS_MR_ASS_FR  (int)26
#define XASS_AR_ASS_FR  (int)28

#define XASS_AC_TMP_FR  (int)17
#define XASS_MR_TMP_FR  (int)18
#define XASS_AR_TMP_FR  (int)20

#define XASS_AC_ASS_NF  (int)9
#define XASS_MR_ASS_NF  (int)10
#define XASS_AR_ASS_NF  (int)12

#define XASS_AC_TMP_NF  (int)1
#define XASS_MR_TMP_NF  (int)2
#define XASS_AR_TMP_NF  (int)4
```

The macro names above are all of the form XASS_1_2_3, where the abbreviations placed at 1, 2, and 3 should be read:

- 1 = AC: always copy
- 1 = MR: may reuse (take pointer if temporary object)
- 1 = AR: always reuse (take pointer)
- 2 = ASS: new object assigned to “variable”
- 2 = TMP: new object temporary
- 3 = FR: call free for old value referred to by variable
- 3 = NF: do not call free for old value

The distinction between all these assignment possibility is only of interest when handling types using or containing pointers.

Generic Equal Functions

Each type in SDL has access to an equal macro `yEqF_typename` and an not equal macro `yNEqF_typename`. Examples for type Boolean and for a user-defined type A:

```
#define yEqF_SDL_Boolean(E1,E2)   ((E1) == (E2))
#define yNEqF_SDL_Boolean(E1,E2)  ((E1) != (E2))

#define yEqF_z3_A(Expr1,Expr2)  yEq_z3_A(Expr1,Expr2)
#define yNEqF_z3_A(Expr1,Expr2)  ( ! yEq_z3_A(Expr1,Expr2) )
#define yEq_z3_A(Expr1,Expr2) \
  GenericEqualSort((void *)Expr1,(void *)Expr2, \
                   (tSDLTypeInfo *)&ySDL_z3_A)
```

These macros are used in the generated code (and in the kernel) at each location where equality tests are needed. The parameters to the equal and not equal macro are the two expressions that should be tested.

If C equal or not equal are not possible to use, the equal macros will become calls to an equal function. The basic generic equal function can be found in `sctpred.h` and `sctpred.h`:

```
extern SDL_Boolean GenericEqualSort(void *, void *,
   tSDLTypeInfo *);
```

where:

- the first two parameters are the addresses to the two expressions to be tested
- the third parameter is the type info node for the actual type.

Special treatment of Charstring and instantiations of the Own generator has made it necessary to introduce specific wrapper functions that in turn calls GenericEqualSort for these types:

```
extern SDL_Boolean xEq_SDL_Charstring
   (SDL_Charstring, SDL_Charstring);
extern SDL_Boolean GenOwn_Equal (void *, void *,
    tSDLTypeInfo *);
```

Generic Free Functions

Each type in SDL that is implemented as a pointer, or that contains a pointer that references to memory that is automatically handled (in principle all pointers except Ref pointers), has access to a corresponding `yFree_typename` function or macro. In the generic function model, this is always a macro.

```
#define yFree_SDL_Charstring(P)  xFree_SDL_Charstring(P)
#define xFree_SDL_Charstring(P) \
   GenericFreeSort(P,(tSDLTypeInfo *)&ySDL_SDL_Charstring)

#define yFree_A(P) \
   GenericFreeSort(P,(tSDLTypeInfo *)&ySDL_A)
```

The yFree macro will always be translated to a call to the function GenericFreeSort.

```
extern void GenericFreeSort (void **, tSDLTypeInfo *);
```

This function takes the address of a variable and a type info node and releases the dynamic memory used by this value contained in the variable.

Generic Make Functions

There are four generic functions constructing values of structured types:

```
extern void * GenericMakeStruct (void *, tSDLTypeInfo *, ...);
extern void * GenericMakeChoice (void *, tSDLTypeInfo *,
   int, void *);
extern void * GenericMakeOwnRef (tSDLTypeInfo *, void *);
extern void * GenericMakeArray  (void *, tSDLTypeInfo *,
   void *);
```

GenericMakeStruct: According to SDL, the Make operator is only available for the struct type. However, in the SDL Suite the Make operator, and thus the GenericMakeStruct function, is also available for the Object_identifier type and the instantiations of the generators string, powerset, and bag.

- The void * parameter is the address of a variable where the result should be placed. This value is also returned.
- The tSDLTypeInfo * parameter is the address to the type info node for the type to be created.
- "..." denotes a list of addresses to the values for the components in the struct. All parameters must be passed as addresses (void *) regardless if the component type should be passed as an address or as a value. The only exceptions are the types represented as pointers themselves (Charstring, Ref, Own, ORef, and syntypes of these types), where the pointers are passed, not the addresses of the pointers. In case of an optional field or a field with an initial value, a '0' or '1' is passed to indicate if a value for the component is present or not. If '1' is passed the value follows as next parameter. If '0' is passed no value is present in the actual parameter list.

GenericMakeChoice: This function is used for choice types.

- The first void * parameter is the address of a variable where the result should be placed. This value is also returned.
- The tSDLTypeInfo * parameter is the address to the type info node for the type to be created.
- The int parameter decides which choice component that is present.

- The last void * parameter is the address of the value.

GenericMakeOwnRef: This function is used for instantiations of generators Own and Ref.

- The tSDLTypeInfo * parameter is the address to the type info node for the type to be created.
- The void * parameter is the address to the value that should be assigned to the memory allocated by this function.

GenericMakeArray: This function is used for instantiations of the generators Array, Carray, and GArray.

- The first void * parameter is the address of a variable where the result should be placed. This value is also returned.
- The tSDLTypeInfo * parameter is the address to the type info node for the type to be created.
- The last void * parameter is the address to the value that should be assigned to all components of the array.

Generic Function for Operators in Pre-defined Generators

The generic function for the operators in the pre-defined generators follow the general rules for operators with a few exceptions:

- a type info node is needed as a parameter, as the C function can handle all instantiations of a certain generator.
- parameters of generator parameter types (component and index types for example) must in many cases be passed as addresses, as the properties of these types are not known.

General array

```
extern void * GenGArray_Extract (xGArray_Type *, void *,
   tSDLGArrayInfo *);
extern void * GenGArray_Modify  (xGArray_Type *, void *,
   tSDLGArrayInfo *);
```

- Parameter 1: The array
- Parameter 2: The index value passed as an address

- Parameter 3: The type info node
- Result: The address of the component

Powerset

Generic functions available for powersets with a simple component type. The powerset is represented a sequences of bits (unsigned char[Appropriate_Length]).

```
#define GenPow_Empty(SDLInfo,Result) \
   memset((void *)Result,0,(SDLInfo)->SortSize)
extern SDL_Boolean GenPow_In (int, xPowerset_Type *,
   tSDLPowersetInfo *);
extern void * GenPow_Incl (int, xPowerset_Type *,
   tSDLPowersetInfo *, xPowerset_Type *);
extern void * GenPow_Del (int, xPowerset_Type *,
   tSDLPowersetInfo *, xPowerset_Type *);
extern void GenPow_Incl2 (int, xPowerset_Type *,
   tSDLPowersetInfo *);
extern void GenPow_Del2 (int, xPowerset_Type *,
   tSDLPowersetInfo *);
extern SDL_Boolean GenPow_LT (xPowerset_Type *,
   xPowerset_Type *, tSDLPowersetInfo *);
extern SDL_Boolean GenPow_LE (xPowerset_Type *,
   xPowerset_Type *, tSDLPowersetInfo *);
extern void * GenPow_And (xPowerset_Type *, xPowerset_Type *,
   tSDLPowersetInfo *, xPowerset_Type *);
extern void * GenPow_Or (xPowerset_Type *, xPowerset_Type *,
   tSDLPowersetInfo *, xPowerset_Type *);
extern SDL_Integer GenPow_Length (xPowerset_Type *,
   tSDLPowersetInfo *);
extern int GenPow_Take (xPowerset_Type *, tSDLPowersetInfo *);
extern int GenPow_Take2 (xPowerset_Type *, SDL_Integer,
   tSDLPowersetInfo *);
```

- **Parameter of type int in GenPow_In, GenPow_Incl, GenPow_Del, GenPow_Incl2, GenPow_Del2**: A component value.
- **Result of type int in GenPow_Take, GenPow_Take2**: A component value.
- **Parameters of type tSDLPowersetInfo ***: The type info node.
- **Parameters of type xPowerset_Type * after the type info node**: The address where the result should be stored. This address is returned by the function.
- **Other xPowerset_Type * parameters**: Powerset in parameters.

Bag and General Powerset

The following generic functions are available for bags and powersets with complex component type. These types are represented as linked lists in C.

```
#define GenBag_Empty(SDLInfo,Result) \
   memset((void *)Result,0,(SDLInfo)->SortSize)
extern void * GenBag_Makebag (void *, tSDLGenListInfo *,
   xBag_Type *);
extern SDL_Boolean GenBag_In (void *, xBag_Type *,
   tSDLGenListInfo *);
extern void * GenBag_Incl (void *, xBag_Type *,
   tSDLGenListInfo *, xBag_Type *);
extern void * GenBag_Del (void *, xBag_Type *,
   tSDLGenListInfo *, xBag_Type *);
extern void GenBag_Incl2 (void *, xBag_Type *,
   tSDLGenListInfo *);
extern void GenBag_Del2 (void *, xBag_Type *,
   tSDLGenListInfo *);
extern SDL_Boolean GenBag_LT (xBag_Type *, xBag_Type *,
   tSDLGenListInfo *);
extern SDL_Boolean GenBag_LE (xBag_Type *, xBag_Type *,
   tSDLGenListInfo *);
extern void * GenBag_And (xBag_Type *, xBag_Type *,
   tSDLGenListInfo *, xBag_Type *);
extern void * GenBag_Or(xBag_Type *, xBag_Type *,
   tSDLGenListInfo *, xBag_Type *);
extern SDL_Integer GenBag_Length (xBag_Type *,
   tSDLGenListInfo *);
extern void * GenBag_Take (xBag_Type *, tSDLGenListInfo *,
   void *);
extern void * GenBag_Take2 (xBag_Type *, SDL_Integer,
   tSDLGenListInfo *, void *);
```

- **Parameter of type int in GenBag_Makebag, GenBag_In, GenBag_Incl, GenBag_Del, GenBag_Incl2, GenBag_Del2**: The address of the component value.
- **Result of type int in GenBag_Take, GenBag_Take2**: The address of the component value.
- **Parameters of type tSDLGenListInfo ***: The type info node.
- **Parameters of type xBag_Type * after the type info node**: The address where the result should be stored. This address is returned by the function.
- **Parameters of type void * after the type info node**: The address where the result should be stored. This address is returned by the function.
- **Other xBag_Type * parameters**: Bag/Powerset in parameters.

String

The following Generic functions are available for String instantiations. A String is implemented as a linked list.

```
#define GenString_Emptystring(SDLInfo,Result) \
   memset((void *)Result,0,(SDLInfo)->SortSize)
extern void * GenString_MkString (void *, tSDLGenListInfo *,
   xString_Type *);
extern SDL_Integer GenString_Length (xString_Type *,
   tSDLGenListInfo *);
extern void * GenString_First (xString_Type *,
   tSDLGenListInfo *, void *);
extern void * GenString_Last (xString_Type *,
   tSDLGenListInfo *, void *);
extern void * GenString_Concat (xString_Type *,
   xString_Type *, tSDLGenListInfo *, xString_Type *);
extern void * GenString_SubString (xString_Type *,
   SDL_Integer, SDL_Integer, tSDLGenListInfo *,
   xString_Type *);
extern void GenString_Append (xString_Type *, void *,
   tSDLGenListInfo *);
extern void * GenString_Extract (xString_Type *, SDL_Integer,
   tSDLGenListInfo *);
```

- **Parameter of type void * in GenString_MkString, GenString_Append**: Address of component value.
- **Parameter of type void * or xString_Type * after type info node**: The address where the result should be stored. This address is returned by the function.
- **Parameters of type tSDLGenListInfo ***: The type info node.
- **Other parameters**: According to SDL definition of parameters.

Limited String

Generic functions available for limited strings, i.e. strings with #STRING directive giving a max size of the string. These strings are implemented as an array in C.

```
#define GenLString_Emptystring(SDLInfo,Result) \
   memset((void *)Result,0,(SDLInfo)->SortSize)
extern void * GenLString_MkString (void *, tSDLLStringInfo *,
   xLString_Type *);
#define GenLString_Length(ST,SDLInfo) (ST)->Length
extern void * GenLString_First (xLString_Type *,
   tSDLLStringInfo *, void *);
extern void * GenLString_Last (xLString_Type *,
   tSDLLStringInfo *, void *);
extern void * GenLString_Concat (xLString_Type *,
   xLString_Type *, tSDLLStringInfo *, xLString_Type *);
extern void * GenLString_SubString (xLString_Type *,
   SDL_Integer, SDL_Integer, tSDLLStringInfo *,
   xLString_Type *);
extern void GenLString_Append (xLString_Type *, void *,
```

```
   tSDLLStringInfo *);
extern void * GenLString_Extract (xLString_Type *,
   SDL_Integer, tSDLLStringInfo *);
```

- **Parameter of type void * in GenLString_MkString, GenString_Append**: Address of component value.
- **Parameter of type void * or xLString_Type * after type info node**: The address where the result should be stored. This address is returned by the function.
- **Parameters of type tSDLLStringInfo ***: The type info node.
- **Other parameters**: According to SDL definition of parameters.

More about Abstract Data Types

Including Type Definitions

> **Note:**
>
> Use the features presented in this section with care. The features were developed early in the SDL Suite history. Now in principle every data type in C can be expressed in SDL as well. Therefore, the recommended method is to write the types in SDL or to translate C types to SDL using the cpp2sdl tool.
>
> History has also shown that it has been difficult to keep full backward compatibility for these features and at the same time improve the performance of the generated code. This of course comes from that these features is highly dependent on the way code is generated.

In this subsection, the inclusion of a type definition in the target language for an abstract data type will be described. When this facility is used, it is necessary to specify how to perform assignment, test for equal, assign default values, and so on, as it is not possible to generate when the type definition is not known (not generated). All this information is given in the #ADT directive, which has the following structure:

```
/*#ADT
  (T(x) A(x) E(x) F(x) K(x) X(x) M(x) W(x) R(x)
   xy 'file name')
#TYPE
C code
#HEADING
C code
#BODY
```

```
C code
*/
```

where each x on the first line should be replaced by one of the characters B, H, Q, S, or G. Replace y by P or I. The interpretation of these characters is similar to the their interpretation for operators.

B	Body
H	Heading
Q	Question
S	Standard
G	Generate
P	Prefix
I	Infix

The reason why G (generate) is not allowed for operators or literals is of course that it would mean to generate the implementation of the operators from the axioms, which is, at least in the general case, an impossible task. For an operator defined in an operator diagram, G is assumed independently of what the user specifies.

The specifications, given in ADT directives, of how to generate code for type definition, assignment, test for equal, default values, and free function should be interpreted according to the table below.

Type Definition

First the actual type definition. The entry - should be interpreted as if no specification is given for T.

Type	Interpretation
T(G)	Generate type definition from SDL sort
T(B)	Do not generate type definition. Assume the type should be “passed as value” to operators
T(BV)	Do not generate type definition. Assume the type should be “passed as address” to operators

Type	Interpretation
T	Same as T(B)
-	Same as T(G)

Assignment

It is possible to select how assignments should be performed for values of the type. Note that all generated assignments will be of the form:

```
yAssF_#(SortName)(....);
```

The yAssF_#(SortName) is a macro either implemented as assignment or as a call to the yAss_#(SortName) function (if such function is to be used), i.e as:

```
#define yAssF_#(S)(V,E,A) V = E
#define yAssF_#(S)(V,E,A) yAss_#(S)(&(V),E,A)
```

Type	Interpretation
A(B)	Use and generate heading, but not body, of yAss_#(S)
A(H)	Use, but generate no code for yAss_#(S)
A(G)	• If the type definition **is** generated: – Use = if possible. – Otherwise use the GenericAssignSort function • If type definition **is not** generated (T, T(B)): – Use =
A(S)	Use =
A	Same as A(B)
-	Same as A(G)

If you define your own assign function, it must be implemented as a function, as the address of the function will be stored (in the type info node for the data type) so GenericAssignSort can call it to handle sub-components of this type. An assign function has the following heading:

```
void yAss_#(SortName)
  (#(SortName) *yVar,
   #(SortName) *yExpr,
   int          AssType)
```

It should assign the value passed as second parameter to the variable passed as first parameter. If the type that is to be assigned contains any pointers the assign function is a bit complicated to write in order to avoid access errors and memory leaks. See the discussion about the AssType parameter to GenericAssignSort in "Generic Assignment Functions" on page 2694.

There is one special case when two assign functions are needed. When the user has decided to write his own assign function and at the same time the type should be passed as value, a second assign function should be added:

```
void yAss2_#(Sortname)
  (#(Sortname) *yVar,
   #(Sortname)  yExpr,
   int   AssType)
```

The difference is that the `yAss2` function takes a value as second parameter, not an address. In generated code the yAss2 function will be used for direct assignment of this type (assignment statements, parameter assignments in input, output, set and so on). The `yAss` function will be called from `GenericAssignSort` to handle assignments of this type, when it is a component of a larger structured type (for example component in struct, array or string).

Equal Test

It is possible to select how test for equality should be performed for values of the type. Note that all generated equal tests will be of the form:

```
yEqF_#(SortName)(....);
```

The yEqF_#(SortName) is a macro either implemented as C equal or as a call to the yEq_#(SortName) function (if such function is to be used), i.e as:

```
#define yEqF_#(S)(E1,E2) E1 == E2
#define yEqF_#(S)(E1,E2) yEq_#(S)(E1,E2)
```

The /= operator is represented by the macro

```
#define yNEqF_#(S)(E1,E2) (! yEqF_#(S)(E1,E2) ).
```

Type	Method
E(B)	Use and generate heading, but not body, of yEq_#(S)
E(H)	Use but generate no code for `yEq_#(S)`
E(G)	• If the type definition **is** generated: – Use `==` if possible – Otherwise use the GenericEqualSort function. • If the type definition **is not** generated – Use `==`
E(S)	Use `==`
E(Q)	Use and generate an equal function that asks for the result of the test (same as Q for operators).
E	Same as E(B)
-	Same as E(G)

If you define your own equal function, it must be implemented as a function, as the address of the function will be stored (in the type info node for the data type) so GenericEqualSort can call it to handle sub-components of this type. An equal function has the following heading:

```
SDL_Boolean yEq_#(SortName)
  (#(SortName) *yExpr1,
   #(SortName) *yExpr2);
```

It should return `true` or `false` depending on if the two values passed as parameters are equal or not. If the parameters contain pointers it might be necessary to free these values, please see the discussion on general parameters to operators in"Other Types Containing Pointers" on page 2690.

Just as for assignment there is a special case when two equal functions are needed. If the user has decided to write his own equal function and at the same time the type should be passed as value, a second equal function should be added:

```
SDL_Boolean yEq2_#(Sortname)
  (#(Sortname) yExpr1,
   #(Sortname) yExpr2)
```

The difference is that the parameters are passed as values instead of as addresses. The `yEq` function will be called from the `GenericEqual-`

`Sort` function to handle when this type is a component in a structured type, while the `yEq2` function is used for direct equal test between values of this type.

Free of Dynamic Memory

This section describes how dynamic memory (if used for the type) will be released for reuse when it is no longer needed.

Type	Interpretation
F(B)	Generate heading, but no body of the free function `yFree_#(SortName)`.
F(H)	Generate neither heading nor body of the free function.
F(S)	Use the function `GenericFreeSort`
F	Same as F(B)
-	Do not use free function.

If you define your own free function, it must be implemented as a function, as the address of the function will be stored (in the type info node for the data type) so GenericFreeSort can call it to handle subcomponents of this type. A free function should have the following prototype

```
void yFree_#(SortName) (void **yVar)
```

The function should take the address to a pointer, return the allocated memory to the pool of available memory and assign 0 to the pointer.

Extract! and Modify!

This entry specifies how component selection (struct components, array components for example) should be performed. In SDL a component can be selected in two ways:

```
Variable ! Component
Variable (Index)
```

An Extract operation can be generated in four ways:

```
Variable.Component       used for struct and #UNIONC
Variable.U.Component     used for #UNION and choice
Variable.A(Index)        used for array
yExtr_SortName(Variable,Expr)
```

The last version, the Extract function, is used for all other cases.

Type	Interpretation
X(B)	Use Extract function
X(G)	Use component selection according to table above.
X	Same as X(B)
-	Same as X(G)

A Modify operation can in the same way be generated in four ways:

```
Variable.Component       used for struct and #UNIONC
Variable.U.Component     used for #UNION and choice
Variable.A(Index)        used for array
(* yAddr_SortName((&Variable), Expr))
```

The last version, the Addr function, is used for all other cases.

Type	Interpretation
M(B)	Use Addr function
M(G)	Use component selection according to table above.
M	Same as M(B)
-	Same as M(G)

Read and Write Function

The write function is used by the monitor system to write values of the type.

Type	Interpretation
W(B)	Generate heading but not the body of a write function.
W(H)	Generate neither heading nor body of a write function, but assume that the user has provided such a function.
W(S)	Values of this type are to be printed as a HEX string. No write function is assumed to be present.
W	Same as W(B)

Type	Interpretation
-	Same as W(S)

The read function is used by the monitor system to read values of the type.

Type	Interpretation
R(B)	Generate heading but not the body of a read function.
R(H)	Generate neither heading nor body of a read function, but assume that the user has provided such a function.
R(S)	Values of this type are to be read as a HEX string. No read function is assumed to be present.
R	Same as R(B)
-	Same as R(S)

In order to examine variable values for variables that are of a sort that is implemented in C, i.e. has #ADT(T) in its ADT directive, it is necessary to implement a write function. Otherwise the value can only be presented as a HEX string. Note that the run-time kernel can automatically handle all SDL sorts for which the code generator generates the C type definition. A write function should look like:

```
extern char * yWri_SortName (void * Value)
```

Given the address of a value of the type SortName, this function should return a char *, i.e. a character string, containing the value represented in a printable form. This character string is the string that will be printed by the monitor, when it needs to print a value of this type. To implement the write function it is not uncommon that a static char array is needed.

Note:

The following two considerations when it comes to write and read functions:

- The read and write functions and any help variables and help functions are surrounded by

```
#ifdef XREADANDWRITEF
#endif
```

This section can be removed if read and write functions are not needed.

- The string format used to represent value of a type should be the same in the read and the write function. **Otherwise communicating simulations will not work** if this type is passed as parameter between the systems.

The function xWriteSort, which is part of the run-time kernel can be useful when implementing Write functions.

```
extern char * xWriteSort (
  void        *In_Addr,
  xSortIdNode  SortNode);
```

The xWriteSort function takes the address of a value to be printed, and a pointer to a xSortIdNode and returns the given value as a string. This function is typically useful if the sort we are implementing a write function for contains one or several components of sorts defined in SDL.

Read Function

In order to assign new values to variables that are of a sort that is implemented in C, i.e. has #ADT(T) in its ADT directive, it is necessary to implement a read function. Otherwise the value can only be read as a HEX string. Note that the run-time kernel can automatically handle all SDL sorts for which the code generator generates the C type definition. A read function should look like:

```
extern int yRead_SortName (void * Result)
```

A read function is given an address to store the value that is read. It should return 1 if the read operation was successful. Otherwise, 0 should be returned and `Result` should be unchanged.

There are some suitable functions in the run-time kernel which can help you when you are implementing a read function. Basically the function xScanToken described below is a tokenizer that transforms sequences of characters to tokens. This function returns tokens according to the following enum type:

```
typedef enum {
  xxId,               /* identifiers, numbers   */
  xxString,           /* SDL Charstring literal */
  xxSlash,            /* /  */
  xxColon,            /* :  */
  xxMinus,            /* -  */
```

```
  xxPlus,              /* +  */
  xxStar,              /* *  */
  xxComma,             /* ,  */
  xxSemicolon,         /* ;  */
  xxArrowHead,         /* ^  */
  xxLPar,              /* (  */
  xxRPar,              /* )  */
  xxLParDot,           /* (. */
  xxRParDot,           /* .) */
  xxLParColon,         /* (: */
  xxRParColon,         /* :) */
  xxDot,               /* .  */
  xxLBracket,          /* [  */
  xxRBracket,          /* ]  */
  xxLCurlyBracket,     /* {  */
  xxRCurlyBracket,     /* }  */
  xxAt,                /* @  */
  xxQuaStart,          /* << */
  xxQuaEnd,            /* >> */
  xxLT,                /* <  */
  xxLE,                /* <= */
  xxGT,                /* >  */
  xxGE,                /* >= */
  xxEQ,                /* =  */
  xxNE,                /* /= */
  xxQuestionMark,      /* ?  */
  xx2QuestionMark,     /* ?? */
  xxExclMark,          /* !  */
  xxSystem,            /* system       */
  xxPackage,           /* package      */
  xxBlock,             /* block        */
  xxProcess,           /* process      */
  xxProcedure,         /* procedure    */
  xxOperator,          /* operator     */
  xxSubstructure,      /* substructure */
  xxChannel,           /* channel      */
  xxSignalroute,       /* signalroute  */
  xxType,              /* type         */
  xxNull,              /* null         */
  xxEnv,               /* env          */
  xxSelf,              /* self         */
  xxParent,            /* parent       */
  xxOffspring,         /* offspring    */
  xxSender,            /* sender       */
  xxEoln,              /* end of line */
  xxEOF,               /* end of file */
  xxErrorToken         /* used to indicate error */
} xxToken;
```

Function xScanToken

The function xScanToken:

```
extern xxToken xScanToken ( char * strVar);
```

reads the next token from input (stdin or Simulator UI) and returns the type of the next token as function result. If the token is xxId or xxString the strVar parameter will contain the identifier, number, or string. The size of the char[] parameter passed as actual parameter should be large enough to store the possible values. If some other token was found, no information is stored in `strVar`.

xUngetToken

Sometimes it is necessary to look-ahead to determine how to handle the current token. Using the function xUngetToken below it is possible return one token to the input. Note that both parameters must have the values obtained from xScanToken.

```
extern void xUngetToken (
  xxToken Token,
  char  * strVar);
```

The functions below can also be useful while implementing Read function. xPromptQuestionMark is suitable to obtain prompt in a similar way as for SDL defined sorts, while xReadOneParameter can be used to read element for element in a list, separated by commas and terminated either by ".)" or "]". The function xReadSort is similar to xWriteSort and can be used to read a component in the treated sort.

xPromptQuestionMark

```
extern xxToken xPromptQuestionMark (
  char  * Prompt,
  char  * QuestionPrompt,
  char  * strVar);
```

The function result and the parameter strVar behave in the same way as for the function xScanToken (see above). The parameter Prompt is the prompt that should be used. This string has to start with a ' ', i.e. a space. To conform with other built-in read function, the Prompt parameter should be: "(SortName) : " (note the ending space colon space). The QuestionPrompt parameter should either be identical to the Prompt parameter, or be null, i.e. (char *)0. If QuestionPrompt is null, the xPromptQuestionMark function will return xxEoln if a end-of-line is found. If QuestionPrompt is not null, the xPromptQuestionMark function will print the QuestionPrompt, and continue to read. Normally QuestionPrompt should be equal to Prompt in a simple data type, while it should be null in a structured data type.

Example 389: ADT Example, Byte Type ––––––––––––––––––––––––

This example is taken from the ADT byte (see "Abstract Data Type for Byte" on page 3254 in chapter 62, *The ADT Library*). The byte type should be read and printed using HEX format.

```
#ifdef XREADANDWRITEF
static char yCTmp[20];

extern int yRead_byte( void  *Result )
{
  unsigned temp;
  xxToken Token;

  Token = xPromptQuestionMark(" (byte) : ",
     " (byte) : ", yCTmp);

  if (Token==xxId && sscanf(yCTmp, "%X", &temp)>=1) {
    *(byte *)Result = (byte)temp;
    return 1;
  }
  xPrintString("Illegal byte value\n");
  return 0;
}

extern char *yWri_byte( void  * C)
{
  sprintf(yCTmp, "%0.2X", *(byte *)C);
  return yCTmp;
}
#endif
```

––

Example 390: ADT Example, Struct Write Functions ––––––––––––

This is an example of how the read and write functions for a struct with two components can look. The monitor system can handle reading of writing of struct values automatically, so please see this just as an example.

```
newtype struct1 /*#NAME 'struct1' */ struct
  a,b Integer;

/*#ADT(W)
#BODY
#ifdef XREADANDWRITEF
static char CTemp[500];

char * yWri_struct1 (void *In_Addr)
{
  strcpy(CTemp, "(. ");
```

```
  strcat(CTemp, xWriteSort((void *)
    (&((struct1 *)In_Addr)->a), xSrtN_SDL_Integer) );
  strcat(CTemp, ", ");
  strcat(CTemp, xWriteSort((void *)
    (&((struct1 *)In_Addr)->b), xSrtN_SDL_Integer) );
  strcat(CTemp, " .)");
  return CTemp;
}
#endif
*/
endnewtype;
```

More about #ADT

When generate is specified for a function, the code generator might decide not to generate the heading of the function, as in some cases it is not needed.

All code that is not generated is assumed to be included by the user in the #TYPE, #HEADING and #BODY sections in the #ADT directive.

Another name for an assign function, equal function and so on may be used, by including the desired name within quotes together with the generation options in the #ADT directive.

If, for example, the name of a certain assign function should be AssX, this can be obtained by specifying: A(B 'AssX') for the assign function. This name will then be used throughout the generated code, both in generated declaration and at the places where the function is called. The name should be last in the specification for the actual function.

An include statement may be generated together with or replacing the type definition by giving a file name within quotes last in the specification part of the #ADT directive, immediately before the first section with code.

Example 391: Including a File in ADT

If the directive

```
/*#ADT (T(B) A(S) E(S) 'file name') */
```

is used, the following include statement will be generated:

```
#include "file name"
```

Note:

Turning off the generation of the objects contained in the include file must be performed by the user.

Directive #REF

Note:

This directive is provided only for backward compatibility. The SDL Suite now supports in/out parameters for operators, which serves exactly the same purpose. In/out parameter is an SDL-2000 extension and is supported also in operator diagrams.

The directive #REF can be used to specify that the **address**, **not the value**, of a variable should be passed as parameter to an ADT operator, as it is defined in SDL. This feature cannot be used for operators defined in operator diagrams (the directive will be ignored for such operators).

The #REF directive is used as shown in the example below.

Example 392: Including a File in ADT

```
operators
  eq1 : Integer, Integer -> Integer;
  eq2 : Integer/*#REF*/, Integer/*#REF*/ -> Integer;
```

The headings for these two operator will become in ANSI-C syntax (ignoring prefixes)

```
extern SDL_Integer eq1 (SDL_Integer P1,
                        SDL_Integer P2);
extern SDL_Integer eq2 (SDL_Integer *P1,
                        SDL_Integer *P2);
```

This feature can be used to optimize parameter passing to operators. The directive, however, also imposes the restriction that the actual parameters must be a variable or a formal parameter (see Example 393 below). This is checked by the code generator. A #REF directive does not in any way effect the way a operator call should be implemented in SDL. It is the responsibility of the code generator to generate the proper actual parameters in C.

Example 393: Including a File in ADT ––––––––––––––––––––––––––

With the ADT in the previous example the following operator call is valid:

```
eq1(sVar, (. 1, 2, 3, 4 .) )
```

The same call of `eq2` would not be valid as the second parameter is not a variable or a formal parameter.

––

Generators

The Cadvanced/Cbasic SDL to C Compiler handles all the predefined generators in SDL, i.e. Array, String, Powerset, and Bag. It is also possible for a user to write his own generators and instantiate them in newtypes. However, the behavior of a user defined generator has to be specified completely by the user. This is performed in a somewhat extended #ADT directive placed just before the endgenerator keyword. These extensions are described below.

There are three additional sections in the directive, apart from #TYPE, #HEADING, and #BODY. These are #INSTTYPE, #INSTHEADING, and #INSTBODY. The inline C code in #TYPE, #HEADING, and #BODY is placed at the point of the generator, i.e. it is generated once. The contents of #INSTTYPE, #INSTHEADING, and #INSTBODY is inserted at each instantiation of the generator, i.e. in each newtype defined using the generator.

In the #INSTTYPE, #INSTHEADING, and #INSTBODY it is possible to use # followed by a number to access the information given in the generator instantiation:

- #0 means the name of the newtype in the instantiation
- #1 and ##1 is the first actual generator parameter
- #2 and ##2 is the second actual generator parameter
- and so on.

#1 and ##1 are equal, except when the corresponding actual generator parameter is a struct (or union). In that case, assuming the SDL struct:

```
newtype aaa struct
  a, b integer;
endnewtype;
```

which will be generated as

```
typedef struct aaa_s {
  SDL_Integer a;
  SDL_Integer b;
} aaa;
```

#1 will become `struct aaa_s` (or `union aaa_s` if a union), while ##1 will become `aaa`.

Example 394: Example of User Defined Generator ––––––––––––––––

```
GENERATOR Ref (TYPE Itemsort)
  LITERALS
    Null;                              /*#OP(S)*/
  OPERATORS
    Ref2VStar : Ref -> VoidStar;       /*#OP(HP)*/
  DEFAULT Null;
/*#ADT()
#INSTTYPE
typedef #1 *#0;
#INSTHEADING
#define #(Null)()        0
#define #(Ref2VStar)(P) ((#(VoidStar))P)
*/
ENDGENERATOR Ref;
```

––

Note the usage of #INSTTYPE and #INSTHEADING in the example above. The code in these section will be inserted in each newtype defined with this generator. For example, in a newtype:

```
NEWTYPE p Ref(Integer)
ENDNEWTYPE;
```

The #INSTTYPE section will become:

```
typedef SDL_Integer *p;
```

Directives to the Cadvanced/Cbasic SDL to C Compiler

Syntax of Directives

The Cadvanced/Cbasic SDL to C Compiler recognizes a number of directives given mainly in SDL comments. The #ADT, #OP, #UNION, and #REF directives used in abstract data types are examples of such directives. The directives #ADT and #OP were described in the section "Implementation of User Defined Operators" on page 2679, "Union" on page 2668, and "Directive #REF" on page 2717, in connection with abstract data types and are not further discussed here.

A directive has the general structure:

1. The start of comment character: /*
2. A '#' character.
3. The directive name.
4. Possible directive parameters given in free syntax. That is, spaces and carriage returns are allowed here.
5. The end of comment characters */.

Upper and lower case letters are considered to be equal in directive names.

Example 395: #OP Directive

Take as an example the directive:

```
/*#OP (B) */.
```

This comment will be recognized as a directive only if no other character is inserted in the sequence `/*#OP`. After this part, spaces and carriage returns may be inserted freely.

Selecting File Structure for Generated Code – Directive #SEPARATE

The purpose of the separate generation feature is to specify the file structure of the generated program. Both the division of the system into a number of files and the actual file names can be specified. There are two ways this information can be given.

- Normally this information is set up in the Organizer, using the *Edit Separation* command, see "Edit Separation" on page 137 in chapter 2, *The Organizer*. Here file names for the generated files can also be specified. In the *Make* dialog in the Organizer (see "Make" on page 120 in chapter 2, *The Organizer*) it is possible to select full separate generation, user-defined separate generation, or no separate generation.

- For an SDL/PR file that is used as input when running the SDL Analyzer as a stand-alone tool, the same information can be entered by #SEPARATE directives directly introduced in the SDL program. Full separate generation, user-defined separate generation, or no separate generation can be set up in the command interface of a stand-alone Analyzer, see *"Set-Modularity" on page 2491 in chapter 54, The SDL Analyzer*.

The Cadvanced/Cbasic SDL to C Compiler can generate a separate file for:

- system (always separate)
- package (always separate)
- system type
- block
- block type
- process
- process type
- service
- service type
- procedure

Note:

Instantiations cannot be separated, i.e. an instance of a block type cannot be generated on a file of its own.

If #SEPARATE directives are used, they should be placed directly after the first semicolon in the system, block, process, or procedure heading; see the following example.

Example 396: #SEPARATE Directive ----------

```
system S; /*#SEPARATE 'filename' */
block B; /*#SEPARATE */
process type P1 inherits PType; /*#SEPARATE */
process P2 (1, ); /*#SEPARATE */
procedure Q; /*#SEPARATE */
```

In the example above the two versions of separate directive, with or without file name, are shown. As can be seen a file name should be enclosed between quotes. The code generator will append appropriate extensions to this name when it generates code.

If no file name is given in the directive, the name of the system, block, process, or procedure will be used to obtain a file name. In such case the file name becomes the name of the unit with the appropriate extension (`.c .h`) depending on contents. The file name is stripped from characters that are not letters, digits or underscores.

The possibility to set up full, user-defined, or no separation in the Organizer's *Make* dialog and in the user interface of a stand-alone Analyzer (see *"The Analyzer Command Line UI" on page 2474 in chapter 54, The SDL Analyzer*), can be used, in simple manner, to select certain default separation schemes. This setting will be interpreted in the following way:

- *No* separation.
  The system and each package will be generated on a separate file.
- *User defined* separation.
  The system, each package, and each unit that the user has specified as separate will become a separate file.
- *Full* separation.
  The system, each package, each block, block type, process, process type, service, and service type will become a separate file. Note that even in this case a procedure is separate only if the user has specified it as separate.

Independently if *No*, *User defined*, or *Full* separation has be selected, the code generator will use the file name specified in the *Edit Separa-*

tion dialog or the #SEPARATE directive, for a file that is to be generated.

An Example of the Usage of the Separate Feature

In the following example a system structure and the #SEPARATE directives are given. The same information can easily be set up in the Organizer as well. This example is then used to show the generated file structure depending on selected generation option.

Example 397: #SEPARATE Directive ––––––––––––––––––––––––––

```
system S; /*#SEPARATE 'Sfile' */
  block B1; /*#SEPARATE */
    process P11; /*#SEPARATE 'P11file' */
    process P12;
  block B2;
    process P21;
    process P22; /*#SEPARATE */
```

Note that #SEPARATE directives can only be used in SDL/PR files. Normally this information is given in the Organizer.

––

Applying Full Separate Generation

If *Full* separate generation is selected then the following files will be generated:

Sfile.c	Sfile.h
B1.c	B1.h
P11file.c	
P12.c	
B2.c	B2.h
P21.c	
P22.c	

The `.c` files contain the C code for the corresponding SDL unit and the `.h` files contain the module interfaces.

Applying Separate Generation

If *User defined* separate generation is selected then the following files will be generated:

`Sfile.c`	`Sfile.h`	Contains code for units S, B2, P21
`B1.c`	`B1.h`	Contains code for units B1, P12
`P11file.c`		Contains code for unit P11
`P22.c`		Contains code for unit P22

The user defined separate generation option thus makes it possible for a user to completely decide the file structure for the generated code. The comments on files and extensions given above are, of course, also valid in this case.

Applying No Separate Generation

If the separation option *No* is selected, only the following file will be generated:

`Sfile.c`		Contains code for all units

The comments on files and extensions earlier are valid even here.

Guidelines

Generally a system should be divided into manageable pieces of code. That is, for a **large system full separate generation** should be used, while for a **small system no separate generation** ought to be used. The possibility to regenerate and recompile only parts of a system usually compensate for the overhead in generating and compiling several files for a large system.

> **Note:**
>
> A file name has to be specified, in the Organizer *Edit Separation* command, see "Edit Separation" on page 137 in chapter 2, *The Organizer*, or in the #SEPARATE directive, if two units in the system have the same name in SDL and should both be generated on separate files, otherwise the same file name will be used for both units.

Accessing SDL Names in C Code – Directive #SDL

When writing C code that is to be included in a generated program it is often necessary to refer to names of objects defined in SDL. The name of an SDL object is, however, transformed when it is translated to C. A prefix, which is a sequence of characters, is added to the SDL name to make the C name unique in the C program. Furthermore, all characters in SDL name which are not allowed in a C name are removed. The prefixes are calculated by looking at the structure of definitions in the actual scope and in all scopes above. This means that adding a declaration at the system level might change all prefixes in blocks and processes contained in the system. As a consequence it is almost impossible to know the prefix of an object in advance.

To be able to write C code and use the name of SDL objects in that code, the Cadvanced/Cbasic SDL to C Compiler provides the directive #SDL which is used in C code to translate an SDL name to the corresponding C name.

The syntax of the #SDL directive is as follows:

```
#SDL (SDL name)
```

or

```
#SDL (SDL name, entity class name)
```

There is also a short form for the directive. No characters are allowed between the # character and the left parentheses in this form:

```
#(SDL name)
```

or

```
#(SDL name, entity class name)
```

Replace `SDL name` with the name of an object in the SDL definition and `entity class name` by any of the following identifiers (upper and lower case letters are considered to be equal):

block blockinst blocksubst blocktype channel channelsubst connect formalpar gate generator label literal newtype	operator package predef procedure process processinst processtype remoteprd remotevar service serviceinst servicetype	signal signallist signalroute sort (= newtype) state synonym syntype system systemtype timer variable view

This list contains all entity classes, which means that not all of the entries are relevant for practical use. When a #SDL directive is found in included C code, the code generator first identifies what SDL object is referred to and then replaces the directive by the C name for that object. The search for the SDL object starts in the current scope (the scope where the C code is included), and follows the scope hierarchy outward to the system definition, until an appropriate SDL object is found. An appropriate SDL name is considered to be found if it has the specified name and is in the specified entity class. If no entity class name is given the search is performed for all entity classes.

Note:

In types, especially in block types, #SDL should be used with care. The reason is that some of the objects in a block type are generated for each instantiation of the block. A #SDL directive on such an object might lead to overloading of names in C. Sensitive objects are processes, process instantiations, signal routes, channels, remote definitions.

The table in the subsection "SDL Predefined Types" on page 2658 gives the direct translation between an SDL name and the corresponding C name or expression. For these names the #SDL directive should not be used.

Escaping # in inline C code

In inline C code # is in some circumstances used for a special meaning.

```
#(name)
#SDL(name)
#0, #1, ..., #9
```

```
##0, ##1, ..., ##9
```

The above constructs are all replaced by the C name for some appropriate unit defined in the source. In all other cases a # will just represent a #. However there are some rare situations when some of the translation rules above makes it impossible to include the intended C code. In format strings it is, for example, possible to have a # followed by a digit. To facilitate this it is possible to escape a # and override the translation rules above.

A sequence of three #, that is ###, will always be copied as one #. The reason for three # is that two # is already used by the translation rules given above.

Example 398: Use of ### in inline code ––––––––––––––––––––––––

Consider the following construction.

```
###1
```

will become:

```
#1
```

––

Including C Code in Task – Directive #CODE

The user's own C code may be included in tasks by using the #CODE directive. This directive has the following syntax:

```
/*#CODE
C code that should
be included in
generated code */
```

Type the directive name on the first line and the C code **on the following lines** up to the end of comment symbol. Note that text on the same line as the #CODE directive are not handled.

A #CODE directive can be placed:

- Immediately preceding or just following the comma that separates two assignment statements or informal texts
- After the last assignment statement or informal text
- Immediately after the ending semicolon (;) of a task (this position is only available in SDL/PR).

The C code in the directives is textually included in the generated code at the position of the directive. If, for example, a code directive is placed between two assignment statements, the code in the directive is inserted between the translated version of the assignment statements.

> **Note:**
>
> The Cadvanced/Cbasic SDL to C Compiler handles the C code in directives as text and performs **no check that the code is valid C code**.

The code directive is included as a facility in the code generator to provide experienced users an escape possibility to the target language C. This increases the application range of the code generator.

An example of a possible use of the code directive is: An algorithm for some computation, which in the SDL description is only indicated as a task with an informal text, could be implemented in C. In this case the directive #SDL described in the previous subsection will probably become useful to access variables and formal parameters defined in SDL. See also the section on "Escaping # in inline C code" on page 2726.

Some general hints on how to write C code that can be included into a simulation program, especially when charstrings or sorts containing charstrings as components are used, can be found in the last part of the section "Implementation of User Defined Operators" on page 2679.

Unfortunately it is not possible to have C comments within the code that is included in any directive, as SDL and C use the same symbols for start and end of comments. See also Example 377 on page 2683 which illustrates the possibility to use the C macro COMMENT.

#CODE directives in compound statements

#CODE directives are recognized after a semicolon that ends a statement of one of the following kinds:

- Output statement
- Create statement
- Set statement
- Reset statement
- Export statement
- Return statement
- Call statement
- Assignment statement
- Break statement
- Continue statement
- Empty statement

Example 399 ----------

```
{
  ;  /*#CODE
       #(i) = #(i)+1; */
  i := i+2;  /*#CODE
       #(i) = #(i)+3; */
}
```

This example contains first an empty statement followed by a directive and then an assignment followed by a directive. The empty statement can, as above, be used to insert #CODE directives at places that otherwise would not be possible, like at the beginning of a compound statement or directly after a compound statement.

Note that the code in the #CODE directive is associated to the statement just before the directive and is included in the scope of that statement.

Example 400 ----------

```
{
  if (true)
    i := i+4; /*#CODE
       #(i) = #(i)+5; */
}
```

In this case the code in the directive is included in the “if-part” of the if statement, just like the assignment it is associated with. This will be treated as:

```
  if (true) {
    i := i+4;
    i := i+5;
  }
```

To put the code directly after the if statement the following structure, with an empty statement after the if statement, can be used:

```
{
  if (true)
    i := i+4;
  ; /*#CODE
       #(i) = #(i)+5; */
}
```

Including C Declarations – Directive #CODE

The #CODE directive can also be used to include C declarations; for example types, variables, functions, #define, and #include in the declaration parts of the C program. This version of the code directive has the following structure:

```
/*#CODE
#TYPE
C code containing:
Types and variables
#HEADING
C code containing:
Extern or static declarations of functions
#BODY
C code containing:
Bodies of functions
*/
```

The separation of functions into `HEADING` and `BODY` sections serves the same purpose as in the #ADT directive, see “Implementation of User Defined Operators” on page 2679.

Code directives to include C declarations may, generally speaking, be placed immediately after a semicolon that ends a declaration in SDL. More precisely it is allowed to place a #CODE directive after the semicolon that ends:

- A heading of a system, block, substructure, process, procedure
- The formal parameters of a process or procedure
- The definition of a block, process, procedure
- The definition of a channel, signal route, signal, signal list, newtype, syntype, synonym, generator, connection, valid input signal set, variable, view, import, timer, remote variable, remote procedure

In the following small PR example the allowed positions are marked with an * followed by a number.

Example 401: #CODE Directive ----------

```
system s; *1
  signal s1, s2(integer); *2
  channel c1 from env to b1
    with s1, s2; *3
  newtype n
    ...
  endnewtype n; *4
  block b1; *5
    signalroute sr1 from env to p1
      with s1, s2; *6
    connect c1 with sr1; *7
    process p1 (1,1); *8
      signalset s1, s2; *9
      dcl a n; *10
      start;
        ...
      state ...;
        ...
    endprocess p1; *11
  endblock b1; *12
endsystem s1;
```

A code directive is considered to belong to the unit where it is defined and the declarations within the directive are thus placed among the other C declaration for that unit. In the example above, directives at positions 1, 2, 3, 4, 12 belong to system s, directives at positions 5,6,7,11 belong to block b1, while directives at positions 8, 9, 10 belong to process p1. Only one code directive may be placed at each available position.

Note:

A variable declared in a #CODE directive that belongs to a process will be shared between the process instances of the process instance set. Such a variable should only be used to represent some common property of all the process instances. To have a variable that is local to a process instance, the variable should be defined in SDL using DCL.

In the generated code the type sections are included in the order of appearance in SDL. However, the type sections are also sensitive for their relative position comparing with SDL sort definitions. This means that the order of the type definitions in the system in the example above will be as follows:

1. Type sections in 1, 2, 3
2. Type generated for newtype n
3. Type sections in 4, 12

As the Cadvanced/Cbasic SDL to C Compiler will generate the SDL sorts in the correct order, definition before usage, in C, the full algorithm is as follows.

- Step through all definitions in SDL in the order of appearance and include:
  - the type section of #CODE directives and #ADT directives.
  - generate typedef for a sort that have no reference to some other not yet generated sort.
- Step through all sorts that have not been generated, checking whether each sort references some other sort that has not been generated. If a sort does not reference some other un-generated sort, it requires typedef generation.
- Repeat the previous step until all sorts have been generated, or until no more sorts can be generated. If sorts remain not generated at this step a recursive dependency has been detected.

The heading sections are placed in the order of their appearance in SDL. This applies to the body sections as well. All body sections will be placed after the sequence of heading sections and the heading section will be placed after all the type definitions. The SDL declarations made in the corresponding unit are available in the code directives and can as usual be reached using the #SDL directive. All declarations made in

code directives are of course available in code directives in tasks in the corresponding unit or in its subunits.

The general hints on how to write C code that fits into a generated C program given in the section "Implementation of User Defined Operators" on page 2679 and in the section "Accessing SDL Names in C Code – Directive #SDL" on page 2725 are also applicable here.

Including C Code in SDL Expressions – Operator #CODE

For each sort defined in an SDL system, both predefined and user defined, the Cadvanced/Cbasic SDL to C Compiler includes an operator #CODE with the following signature:

```
#CODE : Charstring -> S;
```

where S is replaced by the sort name. This operator or rather these operators make it possible to access variables and functions defined in C using the #CODE directive in SDL expressions and still have syntactically and semantically correct SDL expressions.

During code generation, the code generator will just copy the Charstring parameter at the place of the #CODE operator.

Example 402: #CODE Directive ––––––––––––––––––––––––––––––

Suppose that `x` and `y` are SDL variables, which are translated to `z72_x` and `z73_y`, that `a` and `b` are C variables, and `f` is a C function defined in #CODE directives.

SDL expression	C expression
`x + #CODE('a')`	`z72_x + a`
`x + #CODE('a*b')`	`z72_x + a*b`
`x*#CODE('(a+b)')*y`	`z72_x*(a+b)* z73_y`
`#CODE('f(a,#SDL(x))')`	`f(a, z72_x)`

––

Within the Charstring parameter of a #CODE operator the #SDL directive is available in the same way as in other included C code. This is also shown in the last of the examples above.

As there is one #CODE operator for each sort in the system, it is sometimes necessary to qualify the operator with a sort name to make it possible for the SDL Analyzer to resolve which operator that has been used. If, for example, the question and all answers in a decision are given as applications of #CODE operators, then it is not possible to determine the type for the decision. One of the #CODE operators should then be qualified with a sort name to resolve the conflict.

Example 403: Code Directive

```
DECISION #CODE('a');
    (#CODE('1')) : TASK ...;
    (#CODE('2')) : TASK ...;
ENDDECISION;
```

In this case the sort of the decision cannot be resolved. To overcome this problem the question could be written as

```
DECISION TYPE integer #CODE('a');
```

Names and Prefixes in Generated Code

When an SDL name is translated to an identifier in C, a prefix is normally added to the name given in SDL. This prefix is used to prevent name conflicts in the generated code, as SDL has other scope rules than C and also allow different objects defined in the same scope to have the same name, if the objects are of different entity classes. It is, for example, allowed in SDL to have a sort, a variable and a procedure with the same name defined in a process. So the purpose of the prefixes is to make each translated SDL name to a unique name in the C program.

A generated name for an SDL object contains four parts in the following order:

1. The character “z”
2. A sequence of characters that make the name unique. If the object is part of a package, the package name will appear in this sequence.
3. An underscore “_”
4. The SDL name stripped from characters not allowed in C identifiers

Sequence of Characters

A C identifier may contain letters, digits, and underscore “_” and must start with a letter.

The sequence of characters that make the name unique is determined by the position of the declaration in structure of declarations in the system:

- Each declaration on a level is given a number: 0, 1, 2,..., 9, a, b,..., z.

- If the number of declaration on a level is greater than 36, the sequence is: 00, 10, 20, .., 90, a0, .., z0, 01, 11, 21, .., 91, a1, .., z1, ..., 0z, 1z, 2z, .., 9z, az, .., zz.
- If the number of declarations is greater than 36 * 36 then three character sequences are used, and so on.

The total sequence making a name unique is now constructed from the "declaration numbers" for the unit and its parents, that is the units in which it is defined, starting from the top.

If, for example, a sort is defined as the 5th declaration in a block that in turn is the 12th declaration in the system, then the total sequence will be b4 (if not more than 36 declarations are present on any of the two levels).

Example 404: Generated Names in Code ––––––––––––––––––––––

Examples of generated names:

SDL Name	Position of the Declaration	Generated Name
S1	10th declaration in the system	z9_S1
Var2	3rd declaration in the process, which is the 5th declaration in the block, which is the 15th declaration is system	ze42_Var2

––

There will also be other generated names using the prefixes. If, for example, a sort MySort is translated to za2c_MySort, then the equal function connected to this type (if it exists) will be called yEq_za2c_MySort.

Prefixes

> **Note:**
>
> If the OO diagram types in SDL-92 are used (system type, block type, process type), **full prefix should always be used**, as the OO concepts in itself most likely mean the name conflicts will be introduced in C.

This strategy for naming objects in the generated code should be used in all normal situations, as it guarantees that no name conflicts occur. The Cadvanced/Cbasic SDL to C Compiler offers, however, possibilities to change this strategy. In the *Make* dialog in the Organizer (see "Make" on page 120 in chapter 2, *The Organizer*) and in the user interface an Analyzer running stand-alone (see *"The Analyzer Command Line UI" on page 2474 in chapter 54, The SDL Analyzer*), it is possible to select one of the following strategies: *full* prefix, *entity class* prefix, *no* prefix, or *special* prefix. Full prefix is default and is the strategy described above.

Entity Class Prefix

If entity class prefix is selected, then the prefix that is concatenated with the SDL name will be in accordance with the table below and depends only of the entity class of the object.

Entity class	Prefix	Entity class	Prefix
Block, block type, block instance	`blo`	Process, Process type, Process instance	`prs`
Block substructure	`bls`	Remote procedure	`rpc`
Channel	`cha`	Remote variable	`imp`
Channel substructure	`chs`	Service, Service type, Service instance	`ser`
Connection	`con`	Signal	`sig`
Formal parameter	`for`	Signal list	`sil`
Gate	`gat`	Signal route	`sir`
Generator	`gen`	Sort = Newtype	`sor`
Import	`imp`	State	`sta`
Label	`lab`	Syntype	`syt`
Literal	`lit`	Synonym	`syo`
Operator	`ope`	System, System type	`sys`
Package	`pac`	Timer	`tim`
Predef	`pre`	Variable	`var`
Procedure	`prd`	View	`vie`

Using entity class prefix means that the user must guarantee that no name conflict occurs. It also means, however, that the generated names are predictable and thus simplifies writing C code where the SDL names are used. It is only necessary to look for name conflicts within entity classes, for example not having two sorts with the same name. The entity class prefixes handle the case when two objects of different entity class have the same name. Note that the table above contains all entity classes. Not all of the items are actually used by the code generator.

No Prefix

The third alternative, no prefix, means of course that no prefixes are added to the SDL name. The name in the C program will then be the SDL name stripped form characters that are not allowed in C identifiers (everything except letters, digits, and underscore). In this case, the user must guarantee that no name conflict occurs and that the stripped name is allowed as a C identifier, that is, that it begins with a letter.

Special Prefix

In the fourth alternative, special prefix, full prefixes are used for all entity classes except variable, formal parameter, sort, and syntype. For these entity classes no prefix is used.

Conclusion

As was said in the beginning of this subsection, the user should have a good reason for selecting anything but the full prefix, as it could be very difficult to spot name conflicts. The C compiler will in some cases find a conflict, but may in other cases consider the program as legal and generate an executable program with a possibly unwanted behavior. The note above about OO concepts is also a strong argument for full prefix.

Case Sensitivity

Another aspect concerning identifiers is that SDL is case insensitive, while C is case sensitive. The Cadvanced/Cbasic SDL to C Compiler has two translation schemes for identifiers, one is to use the capitalization used in the declaration of the object of concern (default), and one is to use lower case identifiers. The translation scheme is selected in the *Make* dialog in the Organizer (see "Make" on page 120 in chapter 2, *The Organizer*) or in the user interface of the Analyzer, when it is executed stand-alone (*"The Analyzer Command Line UI" on page 2474 in chapter 54, The SDL Analyzer*).

Specifying Names in Generated Code – Directive #NAME

If you wish to decide the name of an object in generated code yourself you can use the #NAME directive. Place the directive directly after the name in the declaration of the object. It should contain the desired name to be used in the generated code within quotes.

Example 405: #NAME Directive

```
NEWTYPE S /*#NAME 'S' */ STRUCT
  a integer;
  b Boolean;
  ADDING OPERATORS
    Op /*#NAME 'OtherName' */ :
      S, S -> Boolean;
ENDNEWTYPE;
```

The name defined in a #NAME directive will be used everywhere that the SDL name is used in the generated code, with two exceptions:

- In the monitor system the SDL names will be used in the communication with the user.
- The name of the files for generated code are not affected by the usage of #NAME directives.

There are, however, some restrictions on where #NAME directives can be placed. Some objects in, for example, a block type are generated in each instantiation of the block type. If a name directive is placed at such an object, the name will probably be overloaded in C, resulting in a C compilation error. The sensitive objects are processes, process instantiations, signal routes, channels, remote variables, and remote procedures.

Assigning Priorities – Directive #PRIO

Priorities can be assigned to processes and process instantiations using the directive #PRIO. The process priorities will affect the scheduling of processes in the ready queue, see "Time" on page 2646. A priority is a positive integer, where low value means high priority. #PRIO directives can be placed directly after the process heading in the definition of the actual process or **last in the reference symbol** (in SDL/GR).

Example 406: #PRIO Directive in process headings ––––––––––––– .

```
Process P1; /*#PRIO 3 */
Process P2(1,1); /*#PRIO 5 */

Process P3 : P3Type; /*#PRIO 3 */
Process P4(1,1) : P4Type; /*#PRIO 5 */
```

––

Processes that do not contain any priority directive will have the default priority 100.

Initialization – Directive #MAIN

The #MAIN directive is used to include initialization code that should be executed before any process transitions are started. The directive should be placed in the system definition directly after the system heading.

In a System diagram the comment with the #MAIN directive should be placed in the additional heading symbol, see "The Additional Heading Symbol" on page 1883 in chapter 43, *Using the SDL Editor*.

Example 407: #MAIN Directive––––––––––––––––––––––––––––––

```
System S;
/*#MAIN
C code for initialization */
```

––

The #MAIN directive has exactly the same structure as the #CODE directive for including code in tasks. The included code will, however, be placed last in the yInit function, after the initialization of the internal structure, but before any transitions are executed.

Modifying Outputs – Directive #EXTSIG, #ALT, #TRANSFER

The purpose of these directives is to modify the standard behavior of an SDL output. The #EXTSIG directive can be used to build applications with the SDL Suite run-time library. The directives #ALT and #TRANSFER are only useful together with other real-time operating systems.

The directive #EXTSIG is used to replace the code for an SDL output with any appropriate in-line C code. This is an optimization and an al-

ternative to the OutEnv function (see chapter 57, *Building an Application*). The #EXTSIG directive can be specified:

- Last in an output symbol (in PR just before the ';').
- Just before the ',' or ';' ending the definition of a signal.

In the first case the #EXTSIG is valid for the signal(s) sent in the output symbol, and in the second case for all outputs of the defined signal.

Example 408: #EXTSIG Directive ----------

```
signal
  Signal1          /*#EXTSIG */,
  Signal2(integer) /*#EXTSIG */;

output Signal3 To Sender /*#EXTSIG */;
```

For each output of a signal with a #EXTSIG directive (in either way described above) the following code is generated:

```
#ifndef EXT_SignalName
  "the normal implementation of an output"
#else
  EXT_SignalName(
    SignalName, ySigN_SignalNameWithPrefix,
    ToExpression, SignalParameters)
#endif
```

where SignalName is the name of the signal in SDL. The parameter ToExpression is a translated version of the SDL expression after TO in the output. If no TO expression is given in the output, this parameter will be xNotDefPId. The entry SignalParameters will be replaced by the list of signal parameter values given in the output.

The intention of this code is to give the user the possibility of introducing a macro with the same name as the signal, where the implementation of the output is expanded to in-line code. By just having a compilation switch which selects if this macro is visible or not, the same generated code can be used both for simulation and for highly optimized applications. An appropriate switch is probably XENV, which governs the normal way of connecting an SDL system to the environment.

Example 409: #CODE Directive ––––––––––––––––––––––––––––––

The following #CODE directive can be included in a text symbol in the system diagram (assuming a signal called SigName with one parameter).

```
/*#CODE
#TYPE
#ifdef XENV
#define EXT_SigName(Name, IdNode, ToExpr, Param1) \
   suitable macro code
#endif
*/
```

––

The other two directives, #ALT and #TRANSFER, can be used together with appropriate real-time operating systems, to have two different interpretations of an output (internal or external output for example) and to specify that a received signal should be immediately retransmitted (#TRANSFER). These kinds of features are not uncommon in real-time operating systems, and can be modeled and simulated by the Cadvanced/Cbasic SDL to C Compiler using these directives. Both these directives should be placed last in the output symbol.

The presence of an #ALT directive will be reflected in the generated code in the way described below.

- If no directive is used, the following macros will be present in generated code for sending a signal:
  - SDL_2OUTPUT: used when the receiver is known.
  - SDL_2OUTPUT_NO_TO: used when the receiver has to be calculated during runtime.
  - SDL_2OUTPUT_COMPUTED_TO: used when the receiver is calculated during code generation.
- If an #ALT directive is given these macros are replaced by:
  - SDL_ALT2OUTPUT
  - SDL_ALT2OUTPUT_NO_TO
  - SDL_ALT2OUTPUT_COMPUTED_TO

In the Master Library, the macros with and without ALT are expanded identically. In an OS integration they might be handled differently to implement two classes of signal sending.

The presence of a #TRANSFER directive indicates that a signal should be directly retransmitted to some other receiver. This can of course be

performed in SDL by an input-output, but then it is necessary to create a new signal and copy the contents of the received signal to the new signal. Using the #TRANSFER directive this copying can be avoided.

In generated code the macros

- ALLOC_SIGNAL
- ALLOC_SIGNAL_PAR

are generated to allocate the data area for a new signal. If a #TRANSFER directive is present in the output statement, these macros are replaced by:

- TRANSFER_SIGNAL
- TRANSFER_SIGNAL_PAR

Note:

In the master library #TRANSFER will still be implemented as a signal copy. It may be possible in an OS integration to avoid the copying if the OS supports such actions.

Normally the #TRANSFER directive should be used in the following way:

```
INPUT signal1(,,,);
OUTPUT signal1(,,,) /*#TRANSFER*/;
```

That is, receive none of the signal parameters in the input and retransmit the signal unchanged. If you want to receive, for example, the second parameter (in variable Var1) and retransmit the signal unchanged except for parameter 3, that should have a new value (73), the following code can be used:

```
INPUT signal1(,Var1,,);
OUTPUT signal1(,,73,) /*#TRANSFER*/;
```

Linking with Other Object Files – Directive #WITH

> **Note:**
>
> This feature is only valid for SDL/PR as input, when the Analyzer is executed stand-alone. Similar features are available in the Organizer's *Make* dialog among the *Generate makefile* options, see "Generate makefile" on page 123 in chapter 2, *The Organizer*.

You can tell the Cadvanced/Cbasic SDL to C Compiler that a number of user defined and precompiled units should be linked together with a generated simulation program. You do this in a #WITH directive that should be placed in the system definition directly after the system heading.

Example 410: #WITH Directive

```
System S; /*#WITH 'file1.o' 'file2.o' */
```

Within the #WITH directive the object files that are to be part in the link operation should be given between quotes, as in the example above. These files will be included in the definition of the link operation in the generated `.m` file.

The make file will, however, not include any definition of how to compile the corresponding source files, as it is impossible for the code generator to know the compilation options or even what compiler the user wants. A user that knows how to interface routines in other languages in a C program, can with this knowledge and the #WITH directive link modules written in another language together with the generated program.

> **Note:**
>
> The #WITH directive will only affect the generated make file. There will be no change in the generated C code when a #WITH directive is introduced.

Naming Tasks in Trace Output – Directive #ID

To simplify the identification of a TASK in a trace printout, the Cadvanced/Cbasic SDL to C Compiler uses the variable on the left hand side of the first assignment statement or the first informal text in the task symbol. A user that is not satisfied with this can name the tasks using #ID directives. An ID directive should contain the character string that is to be used as identification in trace printouts.

Example 411: #ID Directive ––––––––––––––––––––––––––––––––––

```
/*#ID 'Identification of task' */
```

––

The code generator will use the first ID directive it finds in a TASK (if any). An ID directive may be placed:

- First in the task (in PR that is just after the keyword TASK)
- Immediately before or just after a comma separating two assignments or two informal texts
- Last in a task (in PR that is just before the semicolon)
- Directly after the semicolon (this position is only available in SDL/PR).

Directive #C, #SYNT, #SYNTNN, #ASN1

These directives are used to pass information from the tools generating SDL from other languages, for example for C (a `.h` file) or ASN.1.

The #C and the #ASN.1 directive will be inserted after the semicolon ending the package definition or after the package name.

```
package asn1_module; /*#ASN.1 'Module_Name' */
package asn1_module /*#ASN.1 'Module_Name' */;
```

The #ASN.1 directive contains the ASN.1 module name and the #C directive contains the name of the originating `.h` file.

The #SYNT directive and its special form the #SYNTNN directive are used in a package generated from C, to indicate which SDL sorts that are synthesized, i.e. which sorts that where needed in SDL but do not have a name and definition in C. As the originating `.h` file is included in the generated code (with a #include), no typedefs should be generated for the non-#SYNT sorts, while typedefs have to be generated for the synthesized sorts. The #SYNT directive is inserted directly after the name of the newtype or syntype.

Alternative Implementations of the String Generator – Directive #STRING

An instantiation of the string generator can be translated to either a linked list or an array when implemented in C. The #STRING directive is used to determine which translation method to select. The directive should be inserted in a NEWTYPE definition, directly after the string name, when defining a new string data type.

If the #STRING directive is not included in the NEWTYPE definition, the string is implemented as a linked list. This is the default translation method. However, just adding the directive to the NEWTYPE definition is not enough to implement the string as an array. The length of the array must also be defined. This can be done either by using a directive parameter or by using a size constraint in the NEWTYPE definition of the string generator. How this is done is presented in the examples below.

Linked List Implementations

The following examples show how to translate an instantiation of the string generator to a linked list.

Example 412: No #STRING Directive

```
NEWTYPE Example_String
    string(integer, empty)
    constants size (0:10)
ENDNEWTYPE;
```

If the #STRING directive is missing from the NEWTYPE definition, the string will be implemented as a linked list. The length of the list is unlimited, unless a size constraints is defined. In this case the length of the string is within the range of 0 and 10.

Example 413: #STRING Directive without Limited String Size

```
NEWTYPE Example_String /*#STRING */
    string(integer, empty)
ENDNEWTYPE;
```

In this example the string is also implemented as a linked list. The reason for this is that we have not defined a maximum length of the string.

Example 414: #STRING Directive with Parameter Value 0 ––––––––––

```
NEWTYPE Example_String /*#STRING 0*/
    string(integer, empty)
    constants size (0:10)
ENDNEWTYPE;
```

If the parameter value of the #STRING directive is 0, the directive is ignored. Therefore the string will be implemented as a linked list. The size constant will still decide the length of the string.

This example, however, shows how to gradually migrate from a linked list implementation to an array implementation. By just changing the parameter value to anything larger than 0, this NEWTYPE definition creates an array implementation.

––

Array Implementations

The following examples show how to translate an instantiation of the string generator to an array.

Example 415: #STRING Directive with Limited String Size ––––––––

```
NEWTYPE Example_String /*#STRING */
    string(integer, empty)
    constants size (0:10)
ENDNEWTYPE;
```

In this case the string is implemented as an array with the maximum length of 10.

––

Example 416: #STRING Directive with Parameter Value ––––––––––

```
NEWTYPE Example_String /*#STRING 100*/
    string(integer, empty)
    constants size (0:10)
ENDNEWTYPE;
```

In this example the string will be implemented as an array with the length of 100. The parameter value in the #STRING directive overrides the size constant, which in this case is redundant.

––

Size constraint

The size constraint is decided by adding a constant in the NEWTYPE definition. The following declarations are valid when defining a maximum size of the string:

```
constants size (a:b)
constants size (a)
constants size (=a)
constants size (<a)
constants size (<=a)
```

Differences between the Implementations Methods

The different implementation methods affect the behavior and performance of the generated code. The following general statements apply to the methods:

- The performance of the array implementation is usually better than that of the linked list implementation.
- If the number of string values are equal to or close to the maximum string length, the array implementation is smaller.
- If the number of string values are substantially smaller than the maximum string length, the linked list requires less memory.
- The linked list requires memory allocation, while the array does not.

Selecting implementation Methods

It is of course hard to advice which method to select, but the following recommendations apply:

- If the string size is not known, use the linked list.
- If the number of string values is substantially smaller than the maximum string length, use the linked list if speed is not very important.
- If the maximum length is not large or the number of string values are almost equal to the maximum length, use the array.

Using Cadvanced/Cbasic SDL to C Compiler to Generate C++

General

The C code in the Master Library and the C code generated by the Cadvanced/Cbasic SDL to C Compiler is in the common subset of C and C++, and will thus compile both as a C program and as a C++ program. There is one special feature in the code generator concerning C++ when it comes to abstract data types and the possibility to match a C++ class and an SDL data type. Otherwise all the features for including C code, directive #ADT, directive #CODE (see "Abstract Data Types" on page 2656 and "Accessing SDL Names in C Code – Directive #SDL" on page 2725), and so on, are directly applicable for C++ as well. The #CODE directives make it possible to include class definitions as C++ code and the utilization of the classes as C++ code in SDL tasks.

Example 417: Using C++ Classes ––––––––––––––––––––––––––––

CODE directive containing declarations (see "Including C Declarations – Directive #CODE" on page 2730) which should be placed among the SDL declarations:

```
/*#CODE
#HEADING
class TEST {
public:
  void putvar(int avar, int bvar)
    {a = avar; b = bvar;}
  int  geta()
    {return a;}
  int  getb()
    {return b;}
private:
  int a,b;
} TESTvar;
*/
```

Example of usage of the class in a CODE directive in a TASK (see "Including C Code in Task – Directive #CODE" on page 2728).

```
TASK '' /*#CODE
          TESTvar.putvar(#(I), #(I)+10); */;
```

Example of usage of the class in a CODE operators in expressions (see "Including C Code in SDL Expressions – Operator #CODE" on page 2734).

```
OUTPUT Score(
  #CODE('TESTvar.geta()'),
  #CODE('TESTvar.getb()'),
  #CODE('#(TClass)->getVar()')
);
```

Connection Between C++ Classes and SDL

To obtain a close connection between a C++ class and SDL, an abstract data type in SDL can be used; see below.

If you have a C++ class (from a class library or developed specifically for the project), then the following correspondence rules can be used to map the class on an abstract data type.

C++	SDL
Class definition	Abstract data type definition
Class instance pointer	Process variable
Member functions	Operators
new, delete	Operators

Example 418: SDL and C++ Class

Suppose we have a C++ class with the following interface (`.h` file):

```
class TestClass {
public :
  TestClass (int);
  TestClass ();
  ~TestClass ();
  int updateVar(int);
  int getVar();
private :
  int v;
};
```

then the following abstract data type can be used to represent the class (in the example the NAME directive, see "Specifying Names in Generated Code – Directive #NAME" on page 2739, is used to instruct the Cadvanced/Cbasic SDL to C Compiler which name to use in C for particular SDL objects):

```
NEWTYPE TestClass     /*#NAME 'TestClassPtr' */

  LITERALS
```

```
    newTestClass      /*#NAME 'new1TestClass' */
    ;

  OPERATORS
    newTestClass      /*#NAME 'new2TestClass' */
      : integer -> TestClass;

    deleteTestClass   /*#NAME 'deleteTestClass' */
      : TestClass -> TestClass;

    updateVar         /*#NAME 'updateVar' */
      : TestClass, integer -> integer;  /*#OP(HC) */

    getVar            /*#NAME 'getVar' */
      : TestClass -> integer;           /*#OP(HC) */

/*#ADT(T A(S) E(S) D(H) H P)

#TYPE
#include "TestClass.h"
typedef TestClass * TestClassPtr;
COMMENT((NOTE! SDL data type TestClass is
        pointer to C++ class TestClass))

#HEADING
#define yDef_TestClassPtr(p)  *(p) = 0
#define new1TestClass()   new TestClass()
#define new2TestClass(P)  new TestClass(P)

extern TestClassPtr deleteTestClass
  (TestClassPtr);

#BODY
extern TestClassPtr deleteTestClass
  (TestClassPtr P)
{
  delete P;
  return (TestClassPtr)0;
}
*/
ENDNEWTYPE;
```

Note that the #ADT specification means that no code will be generated for the abstract data type. The abstract data type can utilized in the following way:

```
DCL
  TClass TestClass,
  I      Integer;

TASK TClass := newTestClass;
TASK TClass := newTestClass(2);
TASK I := updateVar(TClass, I);
TASK I := getVar(TClass);
```

```
TASK TClass := deleteTestClass(TClass);
```

The only feature that is not described before is the C option in the #OP directive. C (class) is an alternative to I (infix) and P (prefix), and specifies that the operator call should be translated to a member function call of a C++ member function. #OP(C) means that an operator call

```
F(a, b, c)
```

is translated to

```
a->F(b, c)
```

Note:

This means that each operator mapped on class member function should have the class instance pointer as first parameter.

Restrictions

SDL Restrictions

The Cadvanced/Cbasic SDL to C Compiler handles the majority of SDL concepts according to the definition of SDL-92. There are however a number of restrictions that are discussed in this section.

Analyzer Restrictions

The restrictions in the SDL Analyzer are, of course, also valid in the Cadvanced/Cbasic SDL to C Compiler. For more information see "SDL Analyzer" on page 20 in chapter 1, *Compatibility Notes, in the Release Guide*.

Cadvanced/Cbasic SDL to C Compiler Restrictions

The Cadvanced/Cbasic SDL to C Compiler introduces more severe restrictions on the allowed set of SDL concepts than the Analyzer. For more information, see "Cadvanced SDL to C Compiler" on page 24 in chapter 1, *Compatibility Notes, in the Release Guide*.

Migration Guide for Generic Functions

General

This section provides help to migrate a system using old-style code generation for operators to the new Generic Function style.

Introduction

The basic idea of the generic function approach is to decrease the number of generated help functions and functions for operators in predefined generators. This is accomplished by generating generic functions that can be re-used by different types.

This means, for instance, that only one assignment function is created. This function, however, can be used by all types. In the old-style method, one assignment function was created for each type. For operators in predefined generators, there is now **one single** length function calculating the length of **all** string generator instantiations.

In order to implement the generic functions, the parameter passing mechanisms have been changed. In principle, a generic function cannot take a value as parameter, it must receive a pointer to the value. This approach has a positive effect on the performance. However, the generic function approach introduces incompatibility problems if your existing system calls generated functions from inline C-code. If this is the case, the function calls must be changed.

If you need to migrate an SDL system created with the old-style code generation, you must solve the incompatibility issue.

References to Information

Generic Functions

The major source of information regarding generic functions is "Abstract Data Types" on page 2656, which contains a number of sections discussing different aspects of data types and operators:

- "Translation of Sorts" on page 2665 describes how different SDL types are translated to C.

- "Parameter Passing to Operators" on page 2676 discusses the general parameter passing principles for operators and literal functions. This section also lists which types that are passed as values and which types that are passed as addresses.
- "Implementation of User Defined Operators" on page 2679 describes how to include your own implementation in C for an operator.
- "Generic Functions" on page 2693 introduces you to the type info node concept and describes the general operators assign, equal, free, etc.
- "Generic Function for Operators in Pre-defined Generators" on page 2700 describes the operators in the predefined generators.
- "More about Abstract Data Types" on page 2704 comprises information on how to change the implementation of a data type.

SDL Data Types

Information on SDL data types and operators, seen from the SDL point of view, is available in "Using SDL Data Types" on page 42 in chapter 2, *Data Types, in the SDL Suite 6.2 Methodology Guidelines*. Although this section does not discuss the use of generic functions, it provides the framework for SDL data types and operators. In some circumstances it might be better to rewrite a data type or operator in SDL, than to fix the problems in C. A number of extensions and improvements have been included in the support for data types.

Type Info Nodes

For more information on the contents of the type info nodes, please see "Type Info Nodes" on page 3053 in chapter 61, *The Master Library, in the User´s Manual*. This section does not cover migration aspects, but provides implementation details for an interested reader.

Migrating Strategy

Overview

The common problems that might occur when migrating a system from the old-style operator implementation to the generic functions implementation can be divided into two groups:

- Some user-defined SDL operators have changed their prototypes in C. This means that user-provided implementations of the operators have to be changed and that calls to such operators directly from C have to be changed.
- Predefined operators like assignment, equal test, make, and so on, as well as operators in predefined generators might have changed their prototypes in C, which means that calls to such operators directly in C have to be changed. Also types where the implementation of assignment, equal and similar operators are changed by a #ADT directive, might have to be updated.

The first of these problems is fairly straight forward to fix, while the second might be more complex.

Step 1: Identifying Migration Problems

Before continuing with the migration instructions, check if your SDL systems are affected by any incompatibility problems.

1. Find an SDL system that compiles without errors in a previous version of the SDL Suite (with the old-style operator implementations)
2. Open it in the new version of SDL Suite.
3. Compile it and see if you get any compilations errors.

If your compilation errors are similar to the errors in Example 419 on page 2756 you can assume that you have a migration problem.

Example 419: Compilation Errors ––––––––––––––––––––––––––

The compilation error in this example originates from the GNU compiler, gcc. It gives a fairly good feeling of what kind of errors you can expect.

```
file.c:50825: conflicting types for `yAss_example'
file.c:50844: conflicting types for `yEq_example'
file.c:51496: conflicting types for `op1'
file.c:211376: too many arguments to function `op1'
file.c:211376: cannot convert to a pointer type
file.c:6042: incompatible type for argument 2 of
`GenericAssignSort'
file.c:6081: incompatible type for argument 2 of
`op1'
```

––

Step 2: Locating the Compatibility Problems

Note:

Before you try to investigate and correct any error from the list of C compilation errors, you should perform this step. The reason is that many of the errors will point at the wrong place so correcting them might introduce errors rather than correcting anything.

The Cadvanced, Cbasic, and Cmicro SDL to C compilers allows you to find the operators and literals that should be updated. By setting the environment variable `SDT_COMP_WARN` the code generators will produce a file called `compatibility.warn` in the target directory, while generating code for the system. In the first section of this file all operators and literals that might have to be changed are listed.

Example 420: Contents of `compatibility.warn`

```
LITERAL NewDb  C-name: z0V0_NewDb
  <<SYSTEM accesscontrolooa>>
  #SDTREF(TEXT,file.sdl,57,12)
  Literal function result passed as address

OPERATOR ValidateCard  C-name: z0V1_ValidateCard
  <<SYSTEM accesscontrolooa/TYPE CardDbType>>
  #SDTREF(TEXT,file.sdl,59,5)
  Parameter 2 passed as address
    Used in DIRECTIVE at #SDTREF(TEXT,file.sdl,62)
    Used in DIRECTIVE at #SDTREF(TEXT,file.sdl,62)
```

This example shows one literal and one operator that have to be updated:

- The first line contains the name of the operator/literal in SDL and in C.
- The second line contains an appropriate qualifier, that gives information on where in the system the item is defined.
- The third line is the SDT reference to the operator/literal. This can be used in the Organizer's *Goto Source* feature to show where the SDL source is located.
- The following lines indicate where changes are needed. Each line states that the result or a parameter is passed as an address. This means that previously this item was passed as a value, but now it

should be passed as an address. A complete list of types that must be changed can be found in "Mapping Table" on page 2658.

- Last, a number of cross references might be found. These show places where the operator is called from inline C using an #SDL directive. These calls might have to be updated.

Step 3: Updating Operators and Literals

For each operator or literal that is listed in the file `compatibility.warn`, perform the instructions presented in this section.

> **Note:**
> Literals should be treated as operators without parameters.

Updating the Headers

The headers of the corresponding C functions must be updated. If you have specified #OP(B), the header is generated and thus already correct, but if you have specified #OP(H), you have included the header in C probably in the #HEADING section in the #ADT directive for the type.

1. For each parameter that should be passed as an address, add a '*' after the corresponding type name in C.
2. If the result should be passed as an address, add a '*' after the result type in C, and add an extra parameter with the same C type as the updated function result, last among the parameters.

When this step has been performed for all types, the C compilation errors will be reliable again.

Example 421: Headers ----------

```
extern str lit1 (void);
extern str op1 (str, SDL_Integer);
```

Assume `str` is the type that should be passed as an address. The results of both functions and the first parameter of `op1` are mentioned in the `compatibility.warn` file. The heading should be updated to:

```
extern str* lit1 (str*);
extern str* op1 (str*, SDL_Integer, str*);
```

Updating Parameter Specifications

The parameter specification and result in the function implementation must be updated to match the heading. The function implementation is probably in the #BODY section in the #ADT directive of the type.

Example 422: Parameter Specifications (continued from Example 421)

```
str lit1 (void)
{ .... }
str op1 (str P1, SDL_Integer P2)
{ .... }
```

should be changed to:

```
str* lit1 (str* Result)
{ .... }
str* op1 (str* P1, SDL_Integer P2, str* Result);
{ .... }
```

Updating Operator/Literal Functions

The implementation of the operator/literal functions must be updated to reflect the change in parameters.

Example 423: Operator/Literal Functions (continued from Example 422) ----------

In the function `op1` every occurrence of:

`P1` should be replaced by `(*P1)`

Special cases where other changes might be more appropriate:

`&P1` should be replaced by `P1`
`P1.abc` should be replaced by `P1->abc`

The new Result parameter must be assigned the result value of the function and the function must end with a return statement.

Example 424: Return Parameter ----------

```
return Result;
```

It is not unusual that the function contains a local variable used for calculating the result. Normally this variable is no longer needed.

While updating the implementation of the function, information from the next section on, for example, assign and equal function might be valuable.

Note:

Check that parameters that are passed as addresses are NOT CHANGED within the function. If that is the case, copy the value to a local variable first, and work with that variable.

Update Calls to Functions

The calls to the changed functions must be updated. Calls made from SDL cause no problems as the SDL compilers produce the correct code. Only calls made directly from inline C code may have to be updated. The operator list in the file `compatibility.warn` contains cross-references to the inline C code where the operator is called using an `#SDL` directive.

Note:

Operator calls in C without the `#SDL` directive, are not listed in `compatibility.warn`. These calls can only be found via the compiler error list.

For parameters that has changed parameter passing mechanism from pass as value to pass as address you should perform one of the following tasks:

- If the actual parameter is a variable (or something it is possible to take the address of), add a '&', before the variable.
- If the actual parameter is a call to a function, then probably this function has changed its prototype so that it now returns an address. In that case nothing needs to be performed. In other cases proceed to the next possibility.
- If none of the situation above is appropriate, insert a new function local variable of the parameter type, assign the value of the actual parameter to this variable, and insert the address of the variable as actual parameter.

Implementation Hint

> **Note:**
> The following information is not required for the migration of the system, but can be used to improve the performance of the system.

When updating the operators, it might be worth investigating the available features in the SDL Suite, including extensions for operators, in/out parameters, no parameters, no result, etc.

One not too uncommon situation is when a value is passed as an in parameter, then changed by the operator and returned as result value. In every operator call, the same variable is used as both the actual parameter and the receiver of the result. To improve speed of the application, the operator, could be changed to an operator without result and using in/out parameters.

If you perform a change like this, remember to use the cross-reference tool to find all places where the operator is used.

Step 4: Updating typedefs

Overview

Another area where backward incompatibility problems might be present is when the `typedef` is changed in an #ADT directive, especially if the assign, equal, or free functions are changed as well. In the second section of the file `compatibility.warn` all types changing the typedef and at least one of assign, equal, or free functions are listed.

Example 425: Typedef Change ––––––––––––––––––––––––––––––

```
NEWTYPE example  C-name:zDZ_example
  <<SYSTEM mysystem>>
  #SDTREF(TEXT,file.sdl,3096,9)
    Pass as Value  #ADT(T(B)A(B)E(B)F(B))
```

––

The example should be interpreted like this:

- The first line contains the name of the newtype in SDL and in C.
- The second line contains an appropriate qualifier that gives information on where in the system the newtype is defined.

- The third line is the SDT reference to the newtype. This can be used in the Organizer's "goto source" feature to locate the SDL source code.
- The fourth line contains either "Pass as Value" or "Pass as Address", depending on the property of the newtype, followed by the interpretation of the #ADT directive.

As the #ADT directive can be used in many different ways, it is impossible to describe a general method how to correct any problems. The type list in the `compatibility.warn` file indicates what newtypes that have highest probability to cause problems. It is recommended that you go through the listed newtypes and review them given the information in the following sections.

You also have to find and in some cases correct the places where the functions discussed below are called. The compiler error list is the main source of information for this task. Please see "Locating Source Code" on page 2765 if you have problems locating the corresponding SDL source listed in a C compilation error message.

Assignment Functions

For all types passed as addresses, the differences between the old-style, and the new generic assignment functions are:

- The old functions pass the value of the right-hand side expression as the second parameter, while the new pass the address of the right-hand side expression.
- The old functions returns void, but the generic functions return the first parameter, i.e. the address of variable.
- It is now required that the assignment function must be a function (it cannot be a macro), as the address of the function is stored in the type info node for the newtype.

Example 426: The Assignment Function ––––––––––––––––––––––

The prototype for an old assign function would be:

```
void yAss_typename (typename*, typename, int)
```

The prototype for a new assign function should be:

```
typename* yAss_typename (typename*, typename*, int)
```

The body of the assignment function must be updated for these changes.

If assignment for a type passed as an address is used in inline C code, one of the following tasks must be performed:

- If the expression parameter is a variable, a '&', should be added before the variable.
- If the expression is an SDL operator call, a change is normally not needed, as the operator in the generic function model will return the address of a value, not the value itself (for types passed as address).
- You might want to add (void) before the yAss call to tell the compiler that you want to ignore the result.

For more information please see "Generic Assignment Functions" on page 2694.

Equal Functions

For all types passed as addresses, the differences between the old and the new generic equal functions are:

- The old functions pass the values of the two expression, while the new generic equal functions pass the addresses of the expressions.
- It is now required that the equal function must be a function (it cannot be a macro), as the address of the function is stored in the type info node for the newtype.

Example 427: The Equal Function ----------

The prototype for an old equal function would be:

```
SDL_Boolean yEq_typename (typename, typename)
```

The prototype for a new equal function should be:

```
SDL_Boolean yEq_typename (typename*, typename*)
```

The body of the equal function must be updated for these changes.

If equal for a type passed as an address is used in inline C code, perform the same tasks as presented for the assignment function.

For more information please see "Generic Equal Functions" on page 2697.

Free Functions

The `yFree_typename` function is backward compatible. However, it is now required that the equal function must be a function (it cannot be a macro), as the address of the function is stored in the type info node for the newtype.

For more information please see "Generic Free Functions" on page 2698.

Default Functions

The `yDef_typename` macro or function from the non-generic mode is no longer used or generated. Initialization of SDL variables are performed as follows:

- if the variable has an initial value given in the declaration, the variable is assigned this value, using an assignment statement.
- else if the type of the variable has a default value, the variables are assigned this value, using an assignment statement.
- else the variable is set to 0 using memset. If 0 is not an appropriate initial value, the function GenericDefault is called to initialize the variable. Examples of types where 0 is not an appropriate value are structs with components with initial values, Object_identifiers, Strings, general Powersets, Bags and general Arrays.

As yDef functions/macros do not exist in Generic mode, all usage in inline C code must be changed. There are several possibilities. First, decide if assigning a default value is really necessary. If it is necessary, then some of the following principle solutions might be used:

- memset to 0
- ```
  GenericDefault(&variable,
                 (tSDLTypeInfo *)&ySDL_typename)
  ```
- ```
  yAss_typename(variable, expression,
                XASS_MR_ASS_NF)
  ```
- direct C assignment, if possible

Make Functions

The `yMake_typename` function from the non-generic mode is no longer used or generated. All calls in inline code to yMake functions must be replaced by calls to proper GenericMake functions as described in "Generic Make Functions" on page 2699.

> **Note:**
>
> yMake functions pass component values as values, while GenericMake functions pass component values as addresses

Operators in Predefined Generators

The operators in predefined generators have got new generic implementations. All the available generic implementations are listed in "Generic Function for Operators in Pre-defined Generators" on page 2700. Macros that support the old operators names and translate them to the generic functions are generated. If you get compilation errors in such a call you should compare the call with the macro and the generic function, to see if there are any differences in the parameter passing principle.

Locating Source Code

A typical error message from a C compiler lists a file name, a line number, and a description of the error. To locate the corresponding SDL source code perform the following:

1. In a text editor, open the file that is listed in the error message
2. Go to the given line number.
3. Search upwards for an SDT reference, that is, a C comment starting with:

   ```
   /*#SDTREF(
   ```

Copy the SDT reference and paste it into the *Goto Source* feature in the Organizer. The Organizer will show you where the C code originated from.

Chapter 56 The Cadvanced/Cbasic SDL to C Compiler

Chapter

57 *Building an Application*

This chapter describes how you can use the Cadvanced SDL to C Compiler to generate applications and especially how to design the environment functions. These functions allows you connect the SDL system with the environment of the system.

You should read *chapter 56, The Cadvanced/Cbasic SDL to C Compiler* before reading this chapter to understand the general behavior of the Cadvanced SDL to C Compiler. Much of what you need to know to generate an application may be found there, and that information is not repeated here.

Introduction

The Basic Idea

An application generated with the Cadvanced SDL to C Compiler can be viewed as having three parts:

- The SDL system
- The physical environment of the system
- The environment functions, where you connect the SDL system with the environment of the system

In the SDL system process transitions are executed in priority order, signals are sent from one process to another initiating new transitions, timer signals are sent, and so on. These are examples of internal actions that only affect the execution of the SDL system. An SDL system communicates with its environment by sending signals to the environment and by receiving signals from the environment.

The physical environment of an application consists of an operating system, a file system, the hardware, a network of computers, and so on. In this world other actions than just signal sending are required. Examples of actions that an application wants to perform are:

- To read or to write on a file
- To send or receive messages over a network
- To respond on interrupts
- To read and to write information on hardware ports or on sockets

Note:

The entities in the C Code Generator framework, and especially those that use memory allocation, mutexes, semaphores and other synchronization features (implicitly or explicitly) should not be used in interrupt routines, signal handlers or in any similar functions. Depending on the underlying operating system such use may corrupt the application.

The environment functions are the place where the two worlds, the SDL system and the physical environment, meet. Here signals sent from the SDL system to the environment can induce all kinds of events in the physical environment, and events in the environment might cause signals to be sent into the SDL system. You have to provide the environ-

ment functions, as the Cadvanced SDL to C Compiler has no knowledge of the actions that should be performed.

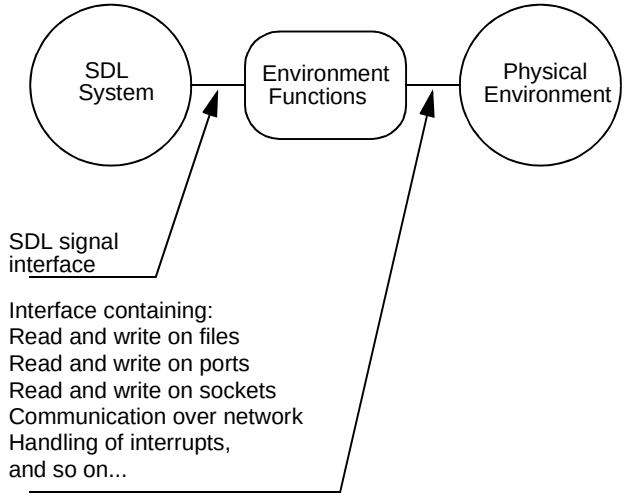


Figure 500: Structure of an application

In a distributed system an application might consist of several communicating SDL systems. Each SDL system will become one executable program. It might execute either as an operating system process, communicating with other operating system processes, or it might execute in a processor of its own, communicating over a network with other processors. There may, of course, also be combinations of these cases. Let us for the sake of simplicity call the operating system processes or processors for nodes communicating over a network. In the case of communicating operating system (OS) processes, the network will be the facilities for process communication provided by the OS.

There are no problems in building an application consisting of several nodes communicating over a network using the Cadvanced SDL to C Compiler. However, you have to implement the communication between the nodes in the environment functions.

Note:

All nodes in a network do not need to be programs generated by the *Cadvanced SDL to C Compiler* from SDL systems. As long as a node can communicate with other nodes, it might be implemented using any technique.

The PId values (references to process instances), are a problem in a distributed world containing several communicating SDL systems. We still want, for example, "Output To Sender" to work, even if Sender refers to a process instance in another SDL system. To cope with this kind of problem, a global node number has been introduced as a component in a PId value. The global node number, which is a unique integer value assigned to each node, identifies the node where the current process instance resides, while the other component in the PId value is a local identification of the process instance within the node (SDL system).

The partitioning of an application into an SDL system and the environment functions has additional advantages. It separates **external actions** into the **logical decision** to perform the action (the decision to send a signal to the environment) and the **implementation details** of the action (treating the signal in the environment functions). This kind of separation reduces the complexity of the problem and allows separate testing. It also allows parallel development of the logic (the SDL system) and the interface towards the environment (the environment functions). When the signal interface between the SDL system and its environment is settled, it is possible to continue both the activities in parallel.

Libraries

Two libraries, *Application* and *ApplicationDebug*, are provided to generate applications. Both use real time (see *"Time" on page 2646 in chapter 56, The Cadvanced/Cbasic SDL to C Compiler* and perform calls to environment functions (see section "The Environment Functions" on page 2774). The difference is that *ApplicationDebug* includes the monitor system while *Application* does not include the monitor system.

When an application is developed, it is usually appropriate to first simulate and debug the SDL system or systems without its environment. One of the libraries *Simulation* or *RealTimeSimulation* may then be used. You first simulate each SDL system on its own and can then simulate the systems together (if you have communicating systems) using

the facility of communicating simulations. After that you probably want to debug the application with the environment functions. This may be performed with the library *ApplicationDebug*. You may then generate the application with the library *Application*.

The library *Validation* allows you to build explorers from the code generated by the Cadvanced SDL to C Compiler. An Explorer has a user interface and executing principles that are similar to a Simulator. The technical approach is however different; an Explorer is based on a technique called *state space exploration* and operates on structures called *behavior trees*. Its purpose is to validate an SDL system in order to find errors and to verify its consistency against Message Sequence Charts.

Reference Section

Representation of Signals and Processes

In this first section, the representation of signals and processes is presented. The symbol table, which is a representation of the static structure of the system, will also be discussed. The information given here will be used in the next part of this section where the environment functions, which should be provided by the user, are described.

Types Representing Signals

A signal is represented by **a C struct** containing **general information** about the signal followed by the **parameters** carried by the signal.

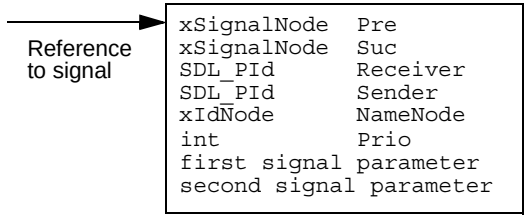


Figure 501: Data structure representing a signal

A general typedef `xSignalRec` for a signal without parameters and for a pointer to such a signal, `xSignalNode`, are given below. These types may be found in the file `scttypes.h`. These types may be used for type casting of any signal to access the general components.

```
typedef struct xSignalRec *xSignalNode;
typedef struct xSignalRec {
  xSignalNode Pre;
  xSignalNode Suc;
  SDL_PId     Receiver;
  SDL_PId     Sender;
  xIdNode     NameNode;
  int         Prio;
} xSignalRec;
```

A `xSignalRec` contains the following components:

- `Pre` and `Suc`. These components are used to link the signal in the input port list of the receiving process instance. The input port is implemented as a double linked list. When a signal has been consumed and the information contained in the signal is no longer needed, the signal will be returned to an avail list to be re-used in future outputs. The component `Suc` is used to link the signal into the avail list, while `Pre` will be `(xSignalNode)0` as long as the signal is in the avail list.
- `Receiver`. The receiving process instance.
- `Sender`. The sending process instance.
- `NameNode`. This component is a pointer to the node in the symbol table that represents the signal type. The symbol table is a tree with information about the SDL system and contains, among other things, one node for each signal type that is defined within the SDL system.
- `Prio`. This component represents the priority of the signal and is used in connection with continuous signals.

In the generated code there will be types to represent the parameters of the signals according to the following example:

Example 428: Generated C Code for Signal Definition ––––––––––––

Assume the following signal definitions in SDL:

```
SIGNAL
 S1(Integer),
 S2,
 S3(Integer, Boolean, OwnType);
```

then the C code below will be generated:

```
typedef struct {
  SIGNAL_VARS
  SDL_Integer Param1;
```

```
} yPDef_z0f_S1;
typedef yPDef_z0f_S1 *yPDP_z0f_S1;

typedef struct {
  SIGNAL_VARS
  SDL_Integer Param1;
  SDL_Boolean Param2;
  z09_OwnType Param3;
} yPDef_z0h_S3;
typedef yPDef_z0h_S3 *yPDP_z0h_S3;
```

where `SIGNAL_VARS` is a macro defined in `scttypes.h` that is expanded to the common components in a signal struct.

For each signal with parameters there are two generated types, a struct type and a pointer type. The struct type contains one component for each parameter in the signal definition and the components will be named `Param1`, `Param2` and so on. The components will be placed in the same order in the struct as the parameters are placed in the signal definition.

> **Note:**
> There are no generated types for a signal without parameters.

Types Representing Processes

A PId value is a struct consisting of two components, a **global node number**, which is an integer (see also "Function xGlobalNodeNumber" on page 2795 and "The Basic Idea" on page 2768) and a **local PId value**, which is a pointer.

```
typedef xLocalPIdRec *xLocalPIdNode;

typedef struct {
  int GlobalNodeNr;
  xLocalPIdNode LocalPId;
} SDL_PId;
```

The global node number identifies the SDL system that the process instance belongs to, while the local PId value identifies the process instance within the system. The local PId pointer value should not be referenced outside the SDL system where it is defined.

By introducing a global node number in the PId values, these values are possible to interpret throughout an application consisting of several

SDL systems. You can also define your own PId values in non-SDL defined parts of the application and still use communication with signals.

The variable `SDL_NULL`, which represents a null value for PIds and which is defined in the runtime library and available through the file `scttypes.h`, contains zero in both the global node number and the local PId component. Note that the global node number should be greater than zero in all PId values except `SDL_NULL`.

The Symbol Table

The symbol table is a tree built up during the initialization phase in the execution of the generated program and contains information about the **static structure of the SDL system**. The symbol table contains, for example, nodes which represent signal types, blocks, channels, process types, and procedures. The C type that are used to represent for example signals in the symbol table is given below.

```
typedef struct xSignalIdStruct *xSignalIdNode;
typedef struct xSignalIdStruct {
 /* components */
} xSignalIdRec;
```

It is the nodes that represent the signal types, for signals sent to and from the environment of the SDL system, that are of major interest in connection with the environment functions. For each signal type there will be a symbol table node. That node may be referenced using the name `ySigN_` followed by the signal name with prefix. Such references may be used in, for example, `xOutEnv` to find the signal type of the signal passed as parameter.

In some cases the symbol table nodes for channels from the environment to a block in the system are of interest to refer to. In a similar way as for signals such nodes may be referenced using the name `yChaN_` followed by the channel name with prefix.

The Environment Functions

An SDL system communicates with its environment by sending signals to the environment and by receiving signals from the environment. As no information about the environment is given in the SDL system, the Cadvanced SDL to C Compiler cannot generate the actions that should be performed when, for instance, a signal is sent to the environment. Instead you have to provide a function that performs this mapping be-

tween a signal sent to the environment and the actions that then should be performed. Examples of such actions are writing a bit pattern on a port, sending information over a network to another computer and sending information to another OS process using some OS primitive.

You should provide the following functions to represent the environment of the SDL system:

- `xInitEnv` and `xCloseEnv`, which are called during initialization and termination of the application
- `xOutEnv` which should treat signals sent to the environment
- `xInEnv` which should treat signals sent into the SDL system from the environment

There are two ways to get a skeleton for the env functions:

- You can copy the file `sctenv.c` from the directory *<installation directory>*`/sdt/sdtdir/`*<machine dependent dir>*`/INCLUDE`
  where <*machine dependent dir*> is for example `sunos5sdtdir` on SunOS 5 and `wini386` in Windows. (**In Windows**, `/` should be replaced by `\` in the path above.)
  This file also contains some trace mechanisms that may be used to trace the execution in a target computer. This trace can, however, only be used if you have the source code for the run-time library (included in the Cadvanced SDL to C Compiler) and can produce a new object library with the appropriate switches.
- You can generate a skeleton by using the *Generate environment functions* option in the Make dialog in the Organizer. In very simple cases you might obtain executable env functions by just tuning the macros in this generated file, but in the general case you must use it as a skeleton and edit it. Remember then to copy the file so that it is not overwritten when code is generated the next time.

An advantage with the generated env functions is that the SDL to C Compiler knows about the signal interface to be implemented in the env functions, and can therefore insert code or macros for all signals in the interface. To calculate this information is not that easy, especially if partitioning (generating code for a part of a system) is used.

Note:

A make template file is generated every time you generate an environment file. This file contains make information for the environment file and possibly for data encoding and decoding files. If you need to change this skeleton file, then remember to copy it so it is not overwritten next time an environment file is generated. The file can be used as make template in the Organizer's generate options. **Note that you may have to generate the file first, before you can select it in the Organizer.**

The env functions are thoroughly discussed below, but first we will introduce the *system interface header file* which considerably simplifies writing the environment functions.

System Interface Header File

The system interface header file contains code for objects that are defined in the system diagram. Included are all type definitions and other external definitions that are needed in order to implement external C code. These object definitions simplify the implementation of the environment functions. Therefore the system interface header file is also known the environment header file. This file is generated if:

- *Code* is generated for the complete system.
- The *Generate environment header file* option is selected in the *Make* dialog in the Organizer (see "Code Generation Options" on page 121 in chapter 2, *The Organizer*).

The default name of the generated interface header file is `<system_file_name>.ifc`.

The system interface header file, has the following structure:

- Macros for all synonyms that are translated to macros.
- All type definitions generated from newtypes and syntypes. This includes `#TYPE` and `#HEADING` sections in `#ADT` directives and in `#CODE` directives.
- External definitions of variables for all synonyms that are translated to variables.

- For each signal defined in the system diagram there will be an extern definition for the `xSignalIdRec` variable representing the signal.
- For each signal with parameters defined in the system diagram, there will be definitions of the types `yPDef_SignalName` and `yPDP_SignalName`, i.e. of the types used to represent a signal.
- For each remote procedure (that can be sent to or from the environment), code will be generated exactly as for two signals named `pCALL_procedurename` and `pREPLY_procedurename`.
- For each channel defined in the system diagram there will be extern definitions for the `xChannelIdRec` representing the channel.

Together with these definitions, macros that simplify the translation of SDL names to C names are also be generated.

Names of SDL Objects in C

Due to differences in naming rules in SDL and C, prefixing is used to make C identifiers unique (see section *"Names and Prefixes in Generated Code" on page 2735 in chapter 56, The Cadvanced/Cbasic SDL to C Compiler)*. These prefixes, however, may change when you update your SDL diagrams and cannot be predicted. Therefore you should not use the prefixed object names in the environment functions. Instead macros, generated in the system interface header file, assist you by mapping static names to the prefixed names. This means that you must regenerate the system interface header file each time you regenerate code for the system. The good part is that you do not have to make any changes in the environment functions, as the interface names are static.

Example 429: Macro in the system interface header file

If an SDL signal called Sig1 is defined in the system, the following macro is created:

```
extern XCONST struct xSignalIdStruct ySigR_z5_Sig1;
#ifndef Sig1
#define Sig1 (&ySigR_z5_Sig1)
#endif
```

This macro allows you to refer to the `xIdNode` by using the static name `Sig1` rather than the prefixed name, `ySigR_z5_Sig1`.

Macros generate static names for the following SDL types:

- Synonyms (both translated to macros and variables).
- Newtypes and Syntypes. If the newtype is translated to an enumeration type, all the literals are available directly in C using their SDL names.
- `xSignalIdNode` representing signals. (No `ySigN_` prefix).
- `xChannelIdNode` representing channels. (Use prefix `xIN_` or `xOUT_` to access the incoming or outgoing direction of the channel).
- The `yPDP_SignalName` pointer type. This type may be referred to using the name `yPDP_SignalName`, where `SignalName` is the SDL name.

Note:

You must always generate the system interface header file before editing or generating the environment functions.

Avoiding name clashes

In SDL it is allowed to give the same name to different objects. This is not allowed in C. For instance, in SDL you can give a signal and a Newtype the same name. In order to distinguish between the names in the system interface header file, you must define static unambiguous names. Using the *Env. Header File Generation* tab in the Targeting Expert, you can do this by using available general identifiers. The identifiers are:

- `%n` - This identifier is the SDL name
- `%s` - This identifier is the SDL name of the scope.
- `sdlobject` - In order to identify the type of object, you can type the object name as a prefix, e.g. signal, literal, etc.

Any combination of the identifiers can be used and they are all optional. However, in order to create a useful system interface header file, it is recommended that the `%n` identifier is always included. Leaving the field empty means that no objects of that type is included in the system interface header file at all.

Note:

If you select to include all objects and use the `%n` identifier only, the system interface header file will become compatible with earlier versions.

Example 430: Name Mapping in an system interface header file

If the signal `Signal1` is in the system `System2` you should type the following in the Signal field in the TAEX.

```
sig_%n_%s
```

The result in the system interface header file will be:

```
sig_Signal1_System2
```

This approach helps you to avoid name clashes in the ifc file. Literals, for example, are often given the same name when defined in different types. The following example shows how this can be solved.

Example 431: Avoiding Name Clashes

In an SDL system the following two newtypes are defined:

```
NewType s
   literals red, green

NewType t
   literals red, yellow
```

As the literal `red` appears in both newtypes, the C code cannot distinguish between them. However, by using the identifiers in the literals field,

```
lit_%n_%s
```

the literals are given the following names:

```
lit_red_s
lit_red_t
```

Thus we will avoid a possible name clash.

Structure of File for Environment Functions

The file containing the environment functions should have the following structure:

```
#include "scttypes.h"
#include "file with macros for external synonyms"
#include "systemfilename.ifc"

void xInitEnv XPP((void))
{
}

void xCloseEnv XPP((void))
{
}

#ifndef XNOPROTO
void xOutEnv (xSignalNode *S)
#else
void xOutEnv (S)
  xSignalNode *S;
#endif
{
}

#ifndef XNOPROTO
void xInEnv (SDL_Time Time_for_next_event)
#else
void xInEnv (Time_for_next_event)
  SDL_Time Time_for_next_event;
#endif
{
}

int xGlobalNodeNumber XPP((void))
{
}
```

The last function, `xGlobalNodeNumber`, will be discussed later, see "Function xGlobalNodeNumber" on page 2795. The usage of the macros `XPP` and `XNOPROTO` makes the code possible to compile both with compilers that can handle prototypes and with compilers that cannot. If you do not need this portability, you can reduce the complexity of the function headings somewhat. In the minor examples in the remaining part of this section, only versions with prototypes are shown.

Functions xInitEnv and xCloseEnv

There are two functions among the environment functions that handle initialization and termination of the environment. These functions, as well as the other environment functions, should be provided by the user.

```
void xInitEnv ( void );

void xCloseEnv ( void );
```

In the implementation of these functions you can place the appropriate code needed to initialize and terminate the software and the hardware. The function `xInitEnv` will be called during the start up of the program as first action, while the `xCloseEnv` will be called in the function `SDL_Halt`. Calling `SDL_Halt` is the appropriate way to terminate the program. The easiest way to call `SDL_Halt` is to include the call in a `#CODE` directive in a TASK. `SDL_Halt` is part of the runtime library and has the following definition:

```
void SDL_Halt ( void );
```

Note:

`xInitEnv` will be called before the SDL system is initialized, which means that no references to the SDL system are allowed in this function. To, for example, send signals into the system during the initialization phase, the `#MAIN` directive should be used (see "Initialization – Directive #MAIN" on page 2740 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*). This directive code can be used after the initialization of the SDL system, but before any transitions are executed.

If you use the generated environment file, you could also define the symbols `XENV_INIT` and `XENV_CLOSE` to be the bodies of `xInitEnv` and `xCloseEnv`.

Function xOutEnv

Each time a signal is sent from the SDL system to the environment of the system, the function `xOutEnv` will be called.

```
void xOutEnv ( xSignalNode *S );
```

The `xOutEnv` function will have the current signal as parameter, so you have all the information contained in the signal at your disposal when you implement the actions that should be performed. The signal con-

tains the signal type, the sending and receiving process instance and the parameters of the signal. For more information about the types used to represent signals and processes, see section "Types Representing Signals" on page 2771 and "Types Representing Processes" on page 2773.

Note that the parameter of `xOutEnv` is an address to `xSignalNode`, that is, an address to a pointer to a struct representing the signal. The reason for this is that the signal that is given as parameter to `xOutEnv` should be returned to the pool of available memory before return is made from the `xOutEnv` function. This is made by calling the function `xReleaseSignal`, which takes an address to an `xSignalNode` as parameter, returns the signal to the pool of available memory, and assigns 0 to the `xSignalNode` parameter. Thus, there should be a call

```
xReleaseSignal(S);
```

before returning from `xOutEnv`. The `xReleaseSignal` function is defined as follows:

```
void xReleaseSignal ( xSignalNode *S );
```

In the generated environment skeleton file, the macro `RELEASE_SIGNAL` releases the signal as described above and returns from xOutEnv.

In the function `xOutEnv` you may use the information in the signal that is passed as parameters to the function. First it is usually suitable to determine the signal type of the signal. This is best performed by `if` statements containing expressions of the following form, assuming the use of the system interface header file and that the signal has the name `Sig1` in SDL:

```
(*S)->NameNode == Sig1
```

The above expression is captured in the macro `IF_OUT_SIGNAL` in the generated environment skeleton file.

Suitable expressions to reach the `Receiver`, the `Sender`, and the signal parameters are:

```
(*S)->Receiver
(*S)->Sender
((yPDP_Sig1)(*S)) -> Param1
((yPDP_Sig1)(*S)) -> Param2
```

(and so on)

Sender will always refer to the sending process instance, while Receiver is either a reference to a process in the environment or the value xEnv. xEnv is a PId value that refers to an environment process instance, which is used to represent the general concept of environment, without specifying an explicit process instance in the environment.

Note:

It is not possible to calculate the PId value for a process in the environment, the value has to be taken from an incoming signal (sender or signal parameter). This is the normal procedure in SDL to establish direct communication between two processes in the same SDL system.

Receiver will refer to the process xEnv if the PId expression in an output TO refers to xEnv, or if the signal was sent in an output without a TO clause and the environment was selected as receiver in the scan for receivers.

Remote procedure calls to or from the environment should in the environment functions be treated a two signals, a pCALL_procedurename and a pREPLY_procedurename signal.

Recommended Structure of the xOutEnv Function

You can, of course, write the xOutEnv function as you wish – the structure discussed below may be seen as an example – but also as a guideline of how to design xOutEnv functions.

Example 432: Structure of xOutEnv Function ----------

```
void xOutEnv ( xSignalNode *S )
{
 if ( (*S)->NameNode == Sig1 ) {
  /* perform appropriate actions */
  xReleaseSignal(S);
  return;
 }
 if ( (*S)->NameNode == Sig2 ){
  /* perform appropriate actions */
  xReleaseSignal(S);
  return;
 }
 /* and so on */
}
```

Function xInEnv

To make it possible to receive signals from the environment and to send them into the SDL system, the user provided function `xInEnv` is repeatedly called during the execution of the system (see section "Program Structure" on page 2796). During such a call you should scan the environment to see if anything has occurred which should trigger a signal to be sent to a process within the SDL system.

```
void xInEnv (SDL_Time Time_for_next_event);
```

To implement the sending of a signal into the SDL system, two functions are available: `xGetSignal`, which is used to obtain a data area suitable to represent the signal, and `SDL_Output`, which sends the signal to the specified receiver according to the semantic rules of SDL. These functions will be described later in this subsection.

The parameter `Time_for_next_event` will contain the time for the next event scheduled in the SDL system. The parameter will either be 0, which indicates that there is a transition (or a timer output) that can be executed immediately, or be greater than Now, indicating that the next event is a timer output scheduled at the specified time, or be a very large number, indicating that there is no scheduled action in the system, that is, the system is waiting for an external stimuli. This large value can be found in the variable `xSysD.xMaxTime`.

You should scan the environment, perform the current outputs, and return as fast as possible if Time has past `Time_for_next_event`.

If Time has not past `Time_for_next_event`, you have a choice to either return from the `xInEnv` function at once and have repeated calls of `xInEnv`, or stay in the `xInEnv` until something triggers a signal output (a signal sent to the SDL system) or until Time has past `Time_for_next_event`.

> **Note:**
>
> We recommend always to return from the `xInEnv` function as fast as possible to ensure that it will work appropriately together with the monitor (during debugging). **Otherwise**, the **keyboard polling**, that is, typing `<RETURN>` in order to interrupt the execution, **will not work**.

The function `xGetSignal`, which is one of the service functions suitable to use when a signal should be sent, returns a pointer to a data area

that represents a signal instance of the type specified by the first parameter.

```
xSignalNode xGetSignal
 ( xSignalIdNode SType,
   SDL_PId Receiver,
   SDL_PId Sender );
```

The components `Receiver` and `Sender` in the signal instance will also be given the values of the corresponding parameters.

- `SType`. This parameter should be a reference to the symbol table node that represents the current signal type. Using the *system interface header file*, such a symbol table node may be referenced using the signal name directly.
- `Receiver`. This parameter should either be a PId value for a process instance within the system, or the value `xNotDefPId`. The value `xNotDefPId` is used to indicate that the signal should be sent as an output without TO clause, while if a PId value is given the output, it is treated as an output with TO clause. Note that PId values for process instances in an SDL system cannot be calculated, but have to be captured from the information (sender or parameter) carried by signals coming from the system. This is the normal procedure in SDL to establish direct communication.
- `Sender`. Sender should either be a PId value representing a process instance in the environment of the current SDL system or the value `xEnv`. `xEnv` is a PId value that refers to an environment process instance, which is used to represent the general concept of the SDL environment, without specifying an explicit process instance in the environment.

The function `SDL_Output` takes a reference to a signal instance and outputs the signal according to the rules of SDL.

```
void SDL_Output
 ( xSignalNode S,
   xIdNode     ViaList[] );
```

- `S`. This parameter should be a reference to a signal instance with all components filled in.
- `ViaList`. This parameter is used to specify if a VIA clause is or is not part of the output statement. The value `(xIdNode *)0` (a null pointer), is used to represent that no VIA clause is present. For information about how to build a via list, please see below.

We now have enough information to be able to write the code to send a signal. Suppose we want to send a signal S1, without parameters, from `xEnv` into the system without an explicit receiver (without TO). The code will then be:

Example 433: C Code to Send a Signal to the Environment ––––––––

```
SDL_Output( xGetSignal(S1, xNotDefPId, xEnv),
  (xIdNode *)0 );
```

If S2, with two integer parameters, should be sent from `xEnv` to the process instance referenced by the variable P, the code will be:

```
xSignalNode OutputSignal; /* local variable */
...
OutputSignal = xGetSignal(S2, P, xEnv);
((yPDP_S2)OutputSignal)->Param1 = 1;
((yPDP_S2)OutputSignal)->Param2 = 2;
SDL_Output( OutputSignal, (xIdNode *)0 );
```

––

For the details of how to reference the parameters of a signal see the subsection "Types Representing Signals" on page 2771.

To introduce a via list in the output requires a variable, which should be an array of `xIdNode`, that contains references to the symbol table nodes representing the current channels (or signal routes) in the via list. In more detail, we need a variable

```
ViaList xIdNode[N];
```

where N should be replaced by the length of the longest via list we want to represent plus one. The components in the variable should then be given appropriate values, such that component 0 is a reference to the first channel (its symbol table node) in the via list, component 1 is a reference to the second channel, and so on. The last component with a reference to a channel must be followed by a component containing a null pointer (the value `(xIdNode)0`). Components after the null pointer will not be referenced. Below is an example of how to create a via list of two channels, C1 and C2.

Example 434: Via List of two Channels.

```
ViaList xIdNode[4];
/* longest via has length 3 */
...
/* this via has length 2 */
ViaList[0] = (xIdNode)xIN_C1;
ViaList[1] = (xIdNode)xIN_C2;
ViaList[2] = (xIdNode)0;
```

The variable `ViaList` may then be used as a `ViaList` parameter in a subsequent call to `SDL_Output`.

Guidelines for the xInEnv Function

It is more difficult to give a structure for the `xInEnv` function, than for the `xOutEnv` function discussed in the previous subsection. A `xInEnv` function will in principle consist of a number of `if` statements where the environment is investigated. When some information is found that means that a signal is to be sent to the SDL system, then the appropriate code to send a signal (see above) should be executed.

The structure given in the example below may serve as an idea of how to design the `xInEnv` function.

Example 435: Structure of xInEnv Function

```
void xInEnv (SDL_Time Time_for_next_event)
{
  xSignalNode S;

  if ( Sig1 should be sent to the system ) {
    SDL_Output (xGetSignal(Sig1, xNotDefPId,
      xEnv), (xIdNode *)0);
  }
  if ( Sig2 should be sent to the system ) {
    S = xGetSignal(Sig1, xNotDefPId, xEnv);
    ((xPDP_Sig2)S)->Param1 = 3;
    ((xPDP_Sig2)S)->Param2 = SDL_True;
    SDL_Output (S, (xIdNode *)0);
  }
  /* and so on */
}
```

This basic structure can be modified to suit your own needs. The if statements could, for example, be substituted for while statements. The signal types might be sorted in some “priority order” and a return can

be introduced last in the if statements. This means that only one signal is sent during a xInEnv call, which reduces the latency.

In the generated environments skeleton file, there are three macros defined to help with allocating and sending a signal;
IN_LOCAL_VARIABLES that declares a signal variable SignalIn,
IN_SIGNAL1 that calls xGetSignal and assigns it to SignalIn, and
IN_SIGNAL2 that calls SDL_Output for SignalIn.

The macros XENV_IN_START and XENV_IN_END can also be defined to be the top and bottom part of the xEnvIn function body.

Alternative to OutEnv - Directive #EXTSIG

To speed up an application it is sometimes possible to use the directive #EXTSIG instead of the xOutEnv function. The decision to use #EXTSIG or xOutEnv may be taken individually for each signal type.

The usage of the #EXTSIG directive is described in the section *"Modifying Outputs – Directive #EXTSIG, #ALT, #TRANSFER" on page 2740 in chapter 56, The Cadvanced/Cbasic SDL to C Compiler*. This information is not repeated here.

By using the #EXTSIG directive the following overhead can be avoided:

- Calling SDL_Output (the library function for outputs)
- SDL_Output determines that the signal is to be sent to the environment
- SDL_Output calls xOutEnv
- xOutEnv executes nested "if" statements to determine the signal type.

Including the Environment Functions in the SDL System Design

Apart from having the environment functions on a file of their own, it is of course possible to include these function directly into the system diagram in a #CODE directive.

Example 436: Including Environment Functions in SDL System ––––

```
/*#CODE
#BODY
```

```
... code for the environment functions ...
*/
```

In this case you cannot use the system interface header file, but instead you have all the necessary declarations already at your disposal, as the functions will be part of the SDL system. The only problem you will encounter is the prefixing of SDL names when they are translated to C. The `#SDL` directive should be used to handle this problem (or the `#NAME` directive), see sections "Accessing SDL Names in C Code – Directive #SDL" on page 2725 and "Specifying Names in Generated Code – Directive #NAME" on page 2739 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*. The following table shows how to obtain C names for some SDL objects of interest:

```
#(Synonym name)
#(Newtype or syntype name)
ySigN_#(Signal name)
yPDP_#(Signal name)
yChaN_#(Channel name)
```

SDL Data Encoding and Decoding, ASCII coder

Communication between nodes often requires data encoding and decoding between node internal representations and a common format in a protocol buffer. The sending node writes information in the common format and the receiving node reads information from the common format. Data encoding is the transformation from a node internal representation into a common format and data decoding is the transformation from a common format into a node internal representation.

Supported common formats are:

- BER
- PER
- ASCII

BER (Basic Encoding Rules) is specified in ITU standard X.690 and PER (Packed Encoding Rules) is specified in ITU standard X.691. BER and PER are based on ASN.1 specifications of data types and can only be used for types specified in ASN.1 specifications. See chapter 58, *ASN.1 Encoding and De-coding in the SDL Suite* and chapter 8, *Tutorial: Using ASN.1 Data Types*.

ASCII is a format where the data is represented as ASCII characters. It is easy to read and analyze. The ASCII format is specified in appendix A.

Example 437 ASCII common format ––––––––––––––––––––––––––

```
newtype Person struct
    nm Charstring;
    nr Integer;
    fm Boolean;
endnewtype Person;

dcl boss Person := (.'Joe',5,false.);

ASCII format:{'Joe',5,F}
```

––

The ASCII coder uses the same buffer management and error management as the BER and PER coders, see chapter 58, *ASN.1 Encoding and De-coding in the SDL Suite*.

Type description nodes for SDL types

SDL data encoders and data decoders need information about types and signals, information that is stored in type descriptions nodes. Type description nodes for an SDL system are generated if the Generate SDL coder option is selected in the organizer, see chapter 2, *The Organizer*, or in the targeting expert, see chapter 59, *The Targeting Expert*. Declarations to access the type nodes are in the system interface header files and in a file with the name `<system_file_name>_cod.h`. Type nodes can be found in any generated c-file from a system or a package and also in the `<system_file_name>_cod.c` file.

A type description node for SDL is implemented as a static variable with the type information. The variable can be accessed by using the name ySDL_<type_name> or ySDL_<signal_name>, where <type_name> and <signal_name> are the names used in the interface header file. See "Names of SDL Objects in C" on page 2778 for more information.

––

Encoding signal and signal parameters into a buffer

You can use an encode function to encode signal parameters into a buffer. There is one encoding function for each common format. For ASCII,

it is accessed by using the macro ASCII_ENCODE. An encoding function has a buffer reference as the first parameter, a pointer to a type node as the second parameter and a pointer to the variable to encode as the third. The encoding function returns an integer value, which is 0 if the encoding was successful and error code if it was not. More details about encoding functions, the buffer reference, type nodes and error codes can be found in chapter 58, *ASN.1 Encoding and De-coding in the SDL Suite*.

Function declarations are in the file `"ascii/ascii.h"` `("ascii\ascii.h" on Windows platforms)` in the coder directory.

Example 438 ASCII encoding of signal parameters ––––––––––––––

```
In SDL:
SIGNAL Sig1(Integer,Person,Boolean);

In xOutEnv:
BufInitWriteMode(buf);
result = ASCII_ENCODE(buf,
              (tSDLTypeInfo *)&ySDL_Integer,
              (void *)&((yPDP_Sig1)(*S))->Param1));
if (result!=0) /* handle error */;
result = ASCII_ENCODE(buf,
              (tSDLTypeInfo *)&ySDL_Person
              (void *)&((yPDP_Sig1)(*S))->Param2));
if (result!=0) /* handle error */;
result = ASCII_ENCODE(buf,
              (tSDLTypeInfo *)&ySDL_Boolean
              (void *)&((yPDP_Sig1)(*S))->Param3));
if (result!=0) /* handle error */;
BufCloseWriteMode(buf);
```

Example 439 ASCII encoding of whole signal (all signal parameters)–

```
In SDL:
SIGNAL Sig1(Integer,Person,Boolean);

In xOutEnv:
BufInitWriteMode(buf);
result = ASCII_ENCODE(buf,
             (tSDLTypeInfo *)&ySDL_Sig1,
              (void *)(*S));
if (result!=0) /* handle error */;
BufCloseWriteMode(buf);
```

Note:

The names of the type nodes in the examples, the second parameter in ASCII_ENCODE, depend on the settings for generating system interface header files.

Decoding into signal parameters from a buffer

You can use a decode function to decode from a buffer into a signal parameter. There is one decoding function for each common format. For ASCII, it is accessed by using the macro ASCII_DECODE. A decode function has a buffer reference as the first parameter, a pointer to a type node as the second parameter and a pointer to the variable to decode as the third. The decoding function returns an integer value, which is 0 if the decoding was successful and an error code if it was not. More details about decoding functions, the buffer reference, type nodes and error codes can be found in chapter 58, *ASN.1 Encoding and De-coding in the SDL Suite*.

Function declarations are in the file `"ascii/ascii.h"` (`"ascii\ascii.h" on Windows platforms`) in the coder directory.

Example 440 ASCII decoding into signal parameters

```
In SDL:
SIGNAL Sig1(Integer,Person,Boolean);

In xInEnv:
BufInitReadMode(buf);
result = ASCII_DECODE(buf,
              (tSDLTypeInfo *)&ySDL_Integer,
              (void *)&((yPDP_Sig1)(S))->Param1));
if (result!=0) /* handle error */;
result = ASCII_DECODE(buf,
              (tSDLTypeInfo *)&ySDL_Person
              (void *)&((yPDP_Sig1)(S))->Param2));
if (result!=0) /* handle error */;
result = ASCII_DECODE(buf,
              (tSDLTypeInfo *)&ySDL_Boolean
              (void *)&((yPDP_Sig1)(S))->Param3));
if (result!=0) /* handle error */;
BufCloseReadMode(buf);
```

Example 441 ASCII decoding of whole signal (all signal parameters)

```
In SDL:
SIGNAL Sig1(Integer,Person,Boolean);
```

```
In xInEnv:
BufInitReadMode(buf);
result = ASCII_DECODE(buf,
              (tSDLTypeInfo *)&ySDL_Sig1,
              (void *)S);
if (result!=0) /* handle error */;
BufCloseReadMode(buf);
```

Note:

The names of the type nodes in the examples, the second parameter in ASCII_DECODE, depend on the settings for generating system interface header files.

Encoding and decoding signal identifier for ASCII encoding

When sending signals between nodes it is often important to put a signal identifier first in the buffer. The signal identifier must be a unique identifier of the signal in the distributed system. Decoding is then a two step process, first decode signal identifier and find signal information and second decode signal parameters.

You can use any representation of signal identifiers in the environment functions.

For SDL types there is a special signal id type node that supports character string signal ids. The type node can be used for ASCII encoding.

Example 442 Using signal identifier

```
In SDL:
SIGNAL Sig1(Integer);

In xOutEnv:
void xOutEnv ( xSignalNode *S )
{
 char * signalId;

 BufInitWriteMode(buf);
 if ( (*S)->NameNode == Sig1 ) {
   /* encode signal id into buffer */
   signalId="Sig1";
   result = ASCII_ENCODE(buf,
                         (tSDLInfo *)&ySDL_SignalId,
                         (void *)signalId);
   if (result!=0) /* handle error */;
   /* encode signal parameter */
   result =  ASCII_ENCODE(buf,
                     (tSDLTypeInfo *)&ySDL_Sig1,
```

```
                        (void *)(*S));
   if (result!=0) /* handle error */;

   /* send buffer using protocol*/

   /* release memory */

   xReleaseSignal(S);
   return;
 }
 BufCloseWriteMode(buf);
}

In xInEnv:
void xInEnv (SDL_Time Time_for_next_event) {
   char SId[100];

   BufInitReadMode(buf);
   /* decode signal id */
   result = ASCII_DECODE(buf,
                    (tSDLTypeInfo *)&ySDL_SignalId,
                     SId) );
   if (result!=0) /* handle error */;
   /* signal Sig1 in buffer */
   if ( strcmp(SId,"Sig1") ) {
      S=xGetSignal(Sig1,xNotDefPId, xEnv);
      result = ASCII_DECODE(buf,
                    (tSDLTypeInfo *)&ySDL_Sig1,
                     (void *)S);
      if (result!=0) /* handle error */;
      SDL_Output(S, (xIdNode *)0 );
   }
   BufCloseReadMode(buf);
}
```

Function xGlobalNodeNumber

You should also provide a function, `xGlobalNodeNumber`, with no parameters, which returns an integer that is unique for each executing system.

```
int xGlobalNodeNumber ( void )
```

The returned integer should be greater than zero and should be unique among the communicating SDL systems that constitutes an application. If the application consists of only one application then this number is of minor interest (it still has to be set). The global node number is used in PId values to identify the node (OS process / processor) that the process instance belongs to. PId values are thereby universally accessible and you may, for example, in a simple way make “Output To Sender” work

between processes in different SDL systems (OS processes / processors).

If you use the generated environment file you could also define `XENV_NODENUMBER` to be the wanted node number.

When an application consisting of several communicating SDL systems is designed, you have to map the global node number to the current OS process or processor, to be able to transmit signals addressed to non-local PIds to the correct OS process or processor. This will be part of the `xOutEnv` function.

Program Structure

The generated code will contain two important types of functions, the initialization functions and the PAD functions. The PAD functions implement the actions performed by processes during transitions. There will be one initialization function in each generated `.c` file. In the file that represents the system this function will have the name `yInit`. Each process in the system will be represented by a PAD function, which is called when a process instance of the current instance set is to execute a transition.

The example below shows the structure of the `main`, `MainInit`, and `MainLoop` functions.

Example 443: Start up structure ––––––––––––––––––––––––––––––

```
void main ( void )
{
 xMainInit();
 xMainLoop();
}

void xMainInit ( void )
{
  xInitEnv();
  Init of internal data structures in the
  runtime library;
  yInit();
}

void xMainLoop ( void )
{
  while (1) {
    xInEnv(...);
    if ( Timer output is possible )
      SDL_OutputTimerSignal();
    else if ( Process transition is possible )
```

```
        Call appropriate PAD function;
    }
}
```

The function `xMainLoop` contains an endless loop. The appropriate way to stop the execution of the program is to call the runtime library function `SDL_Halt`. The call of this C function should normally be included in an appropriate task, using the directive `#CODE`. `SDL_Halt` which has the following structure:

```
void SDL_Halt ( void )
{
  xCloseEnv();
  exit(0);
}
```

To complete this overview, which emphasizes the usage of the environment functions, we have to deal with the `xOutEnv` function. Within PAD functions, the runtime library function `SDL_Output` is called to implement outputs of signals. When `SDL_Output` identifies the receiver of a signal to be a process instance that is not part of the current SDL system, `SDL_Output` will call the `xOutEnv` function.

Dynamic Errors

In the library for applications SDL runtime errors will not be reported. The application will just perform some appropriate actions and continue to execute. These actions are in almost all cases the same as the actions at dynamic errors described in the "Dynamic Errors" on page 2190 in chapter 49, *The SDL Simulator*.

- **Output warnings**: If a signal is sent to NULL or to a stopped process instance, or if no receiver is found in an output without a "to" clause, the signal will not be sent, that is, the output statement is a null action. If a signal is sent to a process instance and there is no path between the sender and the receiver, the signal will be sent anyway (actually, no check will be performed).
- If the error was a **decision error**, that is, no path exists for the current decision value, the execution of the program will continue in an **unpredictable way**. To avoid these kind of problems you should always have else paths in decisions (if not all values in the current data type are covered in other paths).

- If the error occurred during an **import** or **view** action, a data area of the correct size containing zero in all positions is returned.
- **No checks** of **assignment** or **index out of range** will be performed. This means that if an array index is out of bounds, then the corresponding C array will be indexed out of its bounds.
- No checks when accessing struct, #UNION, or choice components are performed. No checks are performed when de-referencing a pointer value (Ref generator). These operations will just be executed.
- If the dynamic error occurred **within an SDL expression**, the operator that found the error will return a default value and the evaluation of the expression is continued. The default values returned depend on the result type of the operator and are given in the section "Default Values" on page 2674 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*.

Example Section

In this section a complete example of an application is presented. The application is simple but it still contains most of the problems that arise when the Cadvanced SDL to C Compiler is used to generate applications. All source code for this example, together with the running application are delivered with the runtime libraries for application generation. **Note that the example is developed for SunOS 5. The example is not updated to use encoding and decoding support.**

The Example

We want to develop an application that consists of several communicating UNIX processes. Each UNIX process should also be connected to the keyboard and the screen. When a complete line is typed on the keyboard (when `<Return>` is pressed) in one of the UNIX processes, that line should be sent to and printed by all the UNIX processes, including the one where the line was entered. If a line starting with the character "." is entered in any UNIX process then all the UNIX processes should terminate immediately.

There are some observations we can make from this short description.

- Firstly, all the UNIX processes should behave exactly the same, which means that the behavior can be described in one SDL system

and one application can be generated. This application should be able to communicate with other instances of itself and should simultaneously be started in as many instances as we want communicating UNIX processes.

- Secondly, each UNIX process needs access to the terminal. To simplify the connection between a UNIX process and the terminal, we assume that each instance of the application is started from its own window (its own shell tool). In this way the underlying window manager will solve the problem of directing input typed on the keyboard to the correct application, as well as making it possible for you to distinguish between output from different applications.
- Thirdly, the UNIX processes have to be connected so they can communicate with each other. We have decided to use sockets as communication media, and to let the UNIX processes form a ring. This means that when a UNIX process receives a message containing a line, it should print the line and send the message on to the next UNIX process in the ring, if it did not itself originally send the line message.

The SDL System

The SDL system with a behavior as outlined above is very simple. It contains, for example, only one process. The system can receive three types of signals, TermInput from the terminal, and Message and Terminate from the SDL system that is the previous node in the ring. The system will respond by sending Display to the terminal and Message and Terminate to the SDL system next in the ring. The signals TermInput and Display take a line (which is read from the terminal or should be printed on the terminal) as parameter. The signal Message takes a line and a PId value (the original sender in the ring) as parameter, while the signal Terminate takes a PId value (the original sender in the ring) as parameter.

The diagrams for the SDL system may be found in “Appendix C: The SDL System” on page 2820. In the section “Where to Find the Example” on page 2807, references to where to find the source code for this example are given.

Simulating the Behavior

At this stage of the development of the application, when the SDL system is completed but the environment functions are not implemented, it is time to simulate the SDL system to debug it at the SDL level. The runtime library *Simulation* is appropriate in this case for simulation.

There are six cases that should be tested:

- If TermInput (with a line not starting with a period) is sent to the system, it should respond by sending a Message signal to the environment. The first parameter in this signal should be equal to the parameter in the received TermInput signal. The second parameter should be the PId value of the sending process.
- If TermInput (with a line starting with a period) is sent to the system, it should respond by sending a Terminate signal to the environment. The parameter should be the PId value of the sending process.
- If Message (with a PId parameter not equal to the receiving process) is sent to the system, it should respond by sending a copy of the Message signal to the environment. It should also send Display to the environment with the received line as parameter.
- If Message (with a PId parameter equal to the receiving process) is sent to the system, it should respond by just sending a Display signal to the environment with the received line as parameter.
- If Terminate (with a PId parameter not equal to the receiving process) is sent to the system, it should respond by sending a copy of the Terminate signal to the environment. The execution of the program should then terminate.
- If Terminate (with a PId parameter equal to the receiving process) is sent to the system, the program should just stop executing.

Let us now verify that the SDL system behaves according to this. In the two executions of the simulation shown below, the cases described above are tested in the same order as they are listed.

Example 444: Execution Trace of Generated Application
Start program `Phone.sim.sct`:

```
Command : set-trace 6
Default trace set to 6
```

```
Command : next-transition

*** TRANSITION START
*      PId    : PhonePr:1
*      State  : start state
*      Now    : 0.0000
*** NEXTSTATE  idle

Command : output-via
Signal name : TermInput
 Parameter 1 (charstring) : 'hello'
Channel name :
Signal TermInput was sent to PhonePr:1 from env:1
Process scope : PhonePr:1

Command : next-transition

*** TRANSITION START
*      PId    : PhonePr:1
*      State  : idle
*      Input  : TermInput
*      Sender : env:1
*      Now    : 0.0000
*      Parameter(s) : 'hello'
*   DECISION  Value: true
*   DECISION  Value: false
*   OUTPUT of Message to env:1
*      Parameter(s) : 'hello', PhonePr:1
*** NEXTSTATE  idle
Command : output-via TermInput '.' -
Signal TermInput was sent to PhonePr:1 from env:1
Process Scope : PhonePr:1

Command : next-transition

*** TRANSITION START
*      PId    : PhonePr:1
*      State  : idle
*      Input  : TermInput
*      Sender : env:1
*      Now    : 0.0000
*      Parameter(s) : '.'
*   DECISION  Value: true
*   DECISION  Value: true
*   OUTPUT of Terminate to env:1
*      Parameter(s) : PhonePr:1
*** NEXTSTATE  idle

Command : output-via Message
 Parameter 1 (charstring) : 'hello'
 Parameter 2 (pid) : env
Channel name :
Signal Message was sent to PhonePr:1 from env:1
Process scope : PhonePr:1
```

```
Command : next-transition

*** TRANSITION START
*       PId     : PhonePr:1
*       State   : idle
*       Input   : Message
*       Sender  : env:1
*       Now     : 0.0000
*       Parameter(s) : 'hello', env:1
*    DECISION  Value: false
*    OUTPUT of Message to env:1
*       Parameter(s) : 'hello', env:1
*    OUTPUT of Display to env:1
*       Parameter(s) : 'hello'
*** NEXTSTATE  idle

Command : output-via Message
 Parameter 1 (charstring) : 'hello'
 Parameter 2 (pid) : PhonePr:1
Channel name :
Signal Message was sent to PhonePr:1 from env:1
Process scope : PhonePr:1

Command : next-transition

*** TRANSITION START
*       PId     : PhonePr:1
*       State   : idle
*       Input   : Message
*       Sender  : env:1
*       Now     : 0.0000
*       Parameter(s) : 'hello', PhonePr:1
*    DECISION  Value: true
*    OUTPUT of Display to env:1
*       Parameter(s) : 'hello'
*** NEXTSTATE  idle

Command : output-via Terminate
 Parameter 1 (pid) : env
Channel name :
Signal Terminate was sent to PhonePr:1 from env:1
Process scope : PhonePr:1

Command : next-transition

*** TRANSITION START
*       PId     : PhonePr:1
*       State   : idle
*       Input   : Terminate
*       Sender  : env:1
*       Now     : 0.0000
*       Parameter(s) : env:1
*    DECISION  Value: false
*    OUTPUT of Terminate to env:1
*       Parameter(s) : env:1
```

```
*   TASK  Halt
```

Example 445 ----------

Start program `Phone.sim.sct`:

```
Command : set-trace 6
Default trace set to 6

Command : next-transition

*** TRANSITION START
*       PId     : PhonePr:1
*       State   : start state
*       Now     : 0.0000
*** NEXTSTATE  idle

Command : output-via Terminate
 Parameter 1 (pid) : PhonePr:1
Channel name :
Signal Terminate was sent to PhonePr:1 from env:1
Process scope : PhonePr:1

Command : next-transition

*** TRANSITION START
*       PId     : PhonePr:1
*       State   : idle
*       Input   : Terminate
*       Sender  : env:1
*       Now     : 0.0000
*       Parameter(s) : PhonePr:1
*   DECISION  Value: true
*   TASK  Halt
```

By running the system with the SDL monitor, as in the examples above, you may debug the system at the SDL level. The overall behavior of the system can thus be tested.

It is possible to start two instances of the simulation and have the simulators communicate with each other. Then Message and Terminate signals sent to the environment in one of the simulations will appear as signals coming from the environment in the other.

> **Note:**
>
> Do not forget the monitor command Start-SDL-Env to make the simulation programs start communicating.

The Environment

In the environment functions we use the socket facility in UNIX to implement the communication between the executing programs. **In the current example, the implementation is developed for SunOS 5.**

To simplify the example we assume that each instance of the application is started in a window of its own (a shell tool window under for instance X Windows, where UNIX commands can be entered). This means that we will have no problems with the interpretation of `stdin` and `stdout` in the programs.

The name of the socket for incoming messages for a certain instance of the application will be the string “Phone” concatenated with the UNIX process number for the current program. The socket will be created in the directory `/tmp`. Each application instance will print this number during the initialization and will then ask for the process number of the application instance where it should send its messages. You have to enter these numbers in such a way as to form a ring among the applications.

The Environment Functions

The environment functions, which may be found in the file `PhoneEnv.c`, are shown in section “Appendix D: The Environment Functions” on page 2823. The file is developed according to the structure discussed in the previous part of this chapter and uses the system interface header file generated from the SDL system.

As the `PhoneEnv.c` file includes `scttypes.h` and uses some C macros, it should be compiled using the same compiler options as the C file for the SDL system. For information about how to extend the generated make file to handle also non-generated files, please see “Makefile Options” on page 123 in chapter 2, *The Organizer*.

In the code for the environment functions a number of UNIX functions are used. Their basic behavior is described below. For any details please see the UNIX manuals available from Sun Microsystems.

Function name	Functionality
`getpid`	Returns the UNIX process number for the current program.
`socket`	Returns a new, unnamed socket.

Function name	Functionality
`bind`	Binds a socket to a name in the file system.
`listen`	Starts listen for other programs trying to connect to this socket.
`connect`	Should be called by other programs that want to establish a connection to the current socket.
`accept`	Accepts a connection request.
`select`	Returns 1 if anything readable can be found in any of the specified file descriptors, where a file descriptor can represent a file, a socket, and the terminal (`stdin` and `stdout`).
`read, write`	Reads or writes on a file (a file descriptor).
`close`	Closes a file.
`unlink`	Removes a file.

If we now look at the code for the environment functions (see "Appendix D: The Environment Functions" on page 2823), we see that `xInitEnv` mainly performs the following actions:

- It creates two unnamed sockets, one for message in (`Connection_Socket`) and one for messages out (`Out_Socket`).
- It binds `Connection_Socket` to the file system (in the `/tmp` directory) and starts listen for connections.
- It prints its UNIX process number.
- It reads the UNIX process number of the process where to send messages.
- It attempts to connect to the socket of the process where to send messages.
- It accepts the connection from the process that will send messages here.

In `xCloseEnv` the created sockets are closed and removed.

The `xInEnv` and `xOutEnv` functions follow the guidelines for these functions given in the reference section. In `xInEnv` the select function

is used to determine if any messages are ready to be received from the terminal (`stdin`) or from the incoming socket. An available message is then read and the information is converted to an SDL signal, which is sent to the SDL system using the `SDL_Output` function. In `xOutEnv` a test on the NameNode in the signal is used to determine the signal type. Depending on the signal type the appropriate information is written either on the outgoing socket or on the terminal (`stdout`).

Debugging

The first part of the debugging activity is, of course, when the SDL system is simulated and examined through the monitor system. Now we also want to include the environment functions during debugging. The intention of the library *ApplicationDebug* is to use the monitor and the environment functions together.

When the environment functions (`xInEnv`) read information from the keyboard there is, however, a problem in using `xInEnv` together with the monitor. In our system, for instance, a line typed on the keyboard may either be a monitor command or a line typed to the SDL system. As both the monitor and `xInEnv` are polling for lines from `stdin`, the interpretation of a typed line depends on which one first finds the line.

A better way is to eliminate this indeterministic behavior by not polling for typed lines in `xInEnv`. Instead, you may use the monitor command:

```
Output-Via TermInput 'the line'
```

to simulate a line typed on the keyboard. In this way all the other parts of the environment functions can be tested under the monitor. If you enclose the sections in `xInEnv` handling keyboard polling between `#ifndef XMONITOR` and `#endif` this code is removed when the monitor is used; that is if the library *ApplicationDebug* is used (see the code for `xInEnv` in "Appendix D: The Environment Functions" on page 2823).

A C source code debugger is of course also useful when debugging the environment functions. The initialization phase, `xInitEnv`, is probably the most difficult part to get working correctly in our system. All the source code for this function is available, and a C debugger can be used.

While debugging generated code from SDL at the C level, it is always easy to find the currently executing SDL symbol, by using the SDT references (see "Syntax" on page 917 in chapter 18, *SDT References*) in the C code and the *Go To Source* menu choice in the *Edit* menu in the

Organizer. For more details please see "Go To Source" on page 100 in chapter 2, *The Organizer*.

Running the Application

To have an application of the Phone system you now only need to make a new executing program with the library *Application*.

When you run the Phone system, start the program from two (or more) shell tools **(on UNIX)**. Each instance of the program will then print:

```
My Pid: 2311
Connect me to:
```

You should answer these questions in such a way that a ring is formed by the programs. When the initialization is completed for a program it prints:

```
******** Welcome to SDT Phone System ********
phone ->
```

The program is now ready to receive lines printed on the keyboard and messages sent from other programs. A Display signal received from another program is printed as follows:

```
display -> the line received in Display signal
```

Where to Find the Example

All files concerning this example may be found in the directory:

```
<installation directory>/sdt/examples/phone
```

Use these files if you only want to look at the source files and if you are using a Sun workstation you could try the executing versions of the program. Otherwise you should copy the files to one of your own directories. Please be sure not to change the original files.

In the directory you will find the following files:

File name	Purpose
`Phone.sdt`	The system file
`Phone.ssy`	Represents the SDL system

File name	Purpose
PhoneBl.sbk	Represents the SDL block
PhonePr.spr	Represents the SDL process
phone.pr	The generated PR file after GR to PR
phone.c	The generated C file after C code generation
phone.ifc	The generated .ifc file
PhoneEnv.c	Contains the environment functions
Phone.solaris.m	The make file for SunOS 5

Appendix A: Formats for ASCII

The ASCII encoding function, ASCII_ENCODE, encodes the SDL signal parameters and variables as ASCII characters before adding into the buffer. The output into the buffer is illustrated with a number of examples.

Braces { } are used for most data types and show where the data types start and stop. Signal parameters start and stop with braces.

Comma is used to delimit.

For more information about data types see "Using SDL Data Types" on page 42 in chapter 2, *Data Types*.

Array, CArray

`Array` is used to define a fixed number of elements.

`Array` takes two generator parameters, an index sort and a component sort.

Example 446: Using Array ----------

```
newtype A1 Array(b, Integer) endnewtype;

dcl Var_Array A1;

task Var_Array := (. 3 .);
```

Output to the buffer: {3,3,3}

Bag

`Bag` is almost the same as `Powerset`. The only difference is that `Bag` contains the same value several times. `Bag` can be used as an abstraction of other data types.

`Bag` takes one generator parameter, the item sort.

Example 447: Using Bag ----------

```
newtype B1 Bag(Integer) endnewtype;

dcl Var_Bag B1;
```

```
task Var_Bag := (. 7, 4, 7 .);
```

Output to the buffer: {2:7,1:4}

Example 448: Using Bag (old-style SDL operator code generation)

```
newtype B1 Bag(Integer) endnewtype;
dcl Var_Bag B1;
task Var_Bag := Incl(7, Incl(4, Incl(7, Empty)));
```

Output to the buffer: {2:7,1:4}

Bit

`Bit` can only take two values, 0 and 1.

Example 449: Using Bit

```
dcl Var_Bit Bit;
task Var_Bit := 1;
```

Output to the buffer: 1

Bit_String

`Bit_String` is used to represent a sequence of bits.

Example 450: Using Bit_String

```
dcl Var_Bit Bit_String;
task Var_Bit := Mkstring(I20(1101));
```

Output to the buffer: '1101'

Boolean

`Boolean` can only take two values, False and True.

Example 451: Using Boolean

```
dcl Var_Boolean Boolean;
```

```
task Var_Boolean := TRUE;
```

Output to the buffer: T

If `Var_Boolean` is set to FALSE, the output to the buffer will be F.

Character

`Character` is used to represent the ASCII characters.

Example 452: Using Character

```
dcl Var_Character Character;

task Var_Character := 'M'
```

Output to the buffer: M

CharStar, PId, UnionC, Userdef, VoidStar, VoidStarStar

The user has to define encoding/decoding procedures for these types, see "Appendix B: User defined ASCII encoding and decoding" on page 2818. If no encoding/decoding procedure is defined, then there will be no output to buffer.

For threaded integrations, the PId value will be encoded/decoded by coding the memory address as an integer.

CharString, IA5String, NumericString, PrintableString, VisibleString

These string types are used to represent sequences of characters.

Example 453: Using Charstring

```
dcl Var_Charstring Charstring;

task Var_Charstring := 'Hello world'
```

Output to the buffer: 'Hello world'

Choice, Union

`Choice` represents the ASN.1 concept CHOICE and can also be seen as a C *union* with an implicit tag field.

Example 454: Using Choice

```
newtype C Choice
  c1 Character;
  c2 Boolean;
endnewtype;

dcl Var_Choice C;

task Var_Choice := c2:true;
```

Output to the buffer: {1,T}, where 1 is an implicit tag.

Duration, Time

`Time` is used to denote ‘a point in time’ and `Duration` is used to denote ‘a time interval’.

A value of sort `Time` represents a point of time in the real world. The `Time` unit is usually 1 second.

Example 455: Using Time

```
dcl Var_Time Time;

task Var_Time := 1;
```

Output to the buffer: {1,0}. The first field is seconds and the second field is nano-seconds.

Enum

`Enum` contains only the values enumerated in a sort.

Example 456: Using Enum

```
newtype E /*Enum*/
  literals e1, e2, e3;
endnewtype;

dcl Var_Enum E;

task Var_Enum := e2;
```

Output to the buffer: 1

Float

`Float` is equivalent to *float* in C.

Example 457: Using Float

```
dcl Var_Float Float;

task Var_Float := 3.14159;
```

Output to the buffer: 3.1415901e0 (always 7 decimals)

GPowerset, Object_Identifier, String

`GPowerset` is used as an abstraction of other data types. `GPowerset` takes one generator parameter, the item sort.

`Object_Identifier` is a sequence of `Natural` values.

`String` can be used to build lists of items of the same type. `String` has two generator parameters, the component sort and the name of an empty string.

Example 458: Using GPowerset

```
newtype GP Powerset(Integer) endnewtype;

dcl Var_GPowerset GP;

task Var_GPowerset := Incl(7, Incl(4, Empty));
```

Output to the buffer: {3,4}

`Object_Identifier` and `String` are used in the same way as `GPowerset`.

Inherits, Syntype

`Syntype` and `Inherits` create a new type, that has the same properties as an existing type, by inheriting the type or make a syntype of the type.

Example 459: Using Syntype

```
syntype newinteger = Integer endsyntype;

dcl Var_Newinteger newinteger;
```

```
task Var_Newinteger := 2
```

Output to the buffer: 2

Integer, LongInt, ShortInt, UnsignedInt, UnsignedLongInt, UnsignedShortInt, Natural

`Integer` is used to represent mathematical integers.

`Natural` is a syntype of `Integer`.

Example 460: Using Integer

```
dcl Var_Integer Integer;

task Var_Integer := 1
```

Output to the buffer: 1

Null

The sort `Null` only contains one value, Null.

Example 461: Using Null

```
dcl Var_Null Null;

task Var_Null := Null;
```

Output to the buffer: 0

Octet

`Octet` is used to represent eight-bit values.

Example 462: Using Octet

```
dcl Var_Octet Octet;

task Var_Octet := I20(12);
```

Output to the buffer: 0c

Octet_String

`Octet_String` represents a sequence of `Octet` values.

`Octet_String` always contains an equal number of characters, since every octet takes two characters.

Example 463: Using Octet_String

```
dcl Var_OctetString Octet_String;

task Var_OctetString := Mkstring(I20(12));
```

Output to the buffer: ‘0c’

ORef, Own, Ref

`ORef, Own` and `Ref` are used to define pointer types.

Example 464: Using Ref

```
newtype R Ref(r1) endnewtype;

dcl Var_Ref R;

task Var_Ref := (. (. 1, 2, ‘Telelogic’ .) .);
```

Output to the buffer: {{1,2,’Telelogic’}}

Powerset

`Powerset` takes one generator parameter, the item sort, and implements a powerset over that sort. `Powerset` can be used as an abstraction of other data types.

Example 465: Using Powerset

```
newtype P Powerset(p1) endnewtype;

dcl Var_Powerset P;

task Var_Powerset := (. 4, 3 .);
```

Output to the buffer: ‘00110000000000000000000000000000’

The bits are an equal multiple of `sizeof(unsigned long)`.

Example 466: Using Powerset (old-style SDL operator code generation)

```
newtype P Powerset(p1) endnewtype;
```

```
dcl Var_Powerset P;

task Var_Powerset := Incl(4, Incl(3, Empty));
```

Output to the buffer: '00110000000000000000000000000000'

The bits are an equal multiple of `sizeof(unsigned long)`.

Real

`Real` is used to represent the mathematical real values.

Example 467: Using Real

```
dcl Var_Real Real;

task Var_Real := 1.0;
```

Output to the buffer: 1.0000000000000e0 (always 13 decimals)

SignalId

`SignalId` is used to describe the signal ID.

`SignalId` is a sequence with characters.

Example 468: Using SignalId

```
dcl Var_SignalId SignalId;

task Var_SignalId := 'Sig1';
```

Output to the buffer: 'Sig1'

Struct

`Struct` can be used to make an aggregate of data that belong together.

Example 469: Using Struct

```
newtype S struct
  s1 integer;
  s2 integer;
  s3 charstring;
endnewtype;
```

```
dcl Var_Struct S;

task Var_Struct := (. 1, 2, 'Telelogic' .);
```

Output to the buffer: {1,2,'Telelogic'}

Appendix B: User defined ASCII encoding and decoding

The types CharStar, PId, UnionC, Userdef, VoidStar, VoidStarStar do not have a natural transformation to ASCII format. The desired encoding probably differs from application to application. The ASCII encoding procedure and the ASCII decoding procedures do not encode or decode these types, but they can invoke user written procedures for encoding and decoding them.

For threaded integrations, the PId value will be encoded/decoded by coding the memory address as an integer.

If you want to add encoding for these types, then do the following steps:

- Implement a C-function for encoding with input and output parameters compatible with tEncodeFunc, which is declared in file “coderucf.h”. The encode functions in “coderascii.c” can be used as an example.
- Set static variable AsciiUserEncode to your encode function in xInitEnv.

If you want to add decoding for these types, then do the following steps:

- Implement a C-function for decoding with input and output parameters compatible with tDecodeFunc, which is declared in file “coderucf.h”. The decode functions in “coderascii.c” can be used as an example.
- Set static variable AsciiUserDecode to your decode function in xInitEnv.

Example 470 User defined ASCII encoding ––––––––––––––––––––

```
int MyAsciiEncoder( tBuffer Buf,
                    tSDLTypeInfo* TypeNode,
                    void* Value )
{
  /* my error handling code for one or more of
     CharStar, PId, UnionC, Userdef, VoidStar, VoidStarStar */
  /* return 1 if it was succesful,
     return 0 if it failed */
}

In xInitEnv:
```

```
  AsciiUserEncode = MyAsciiEncoder;
-------------------------------------------------------
```

Appendix C: The SDL System

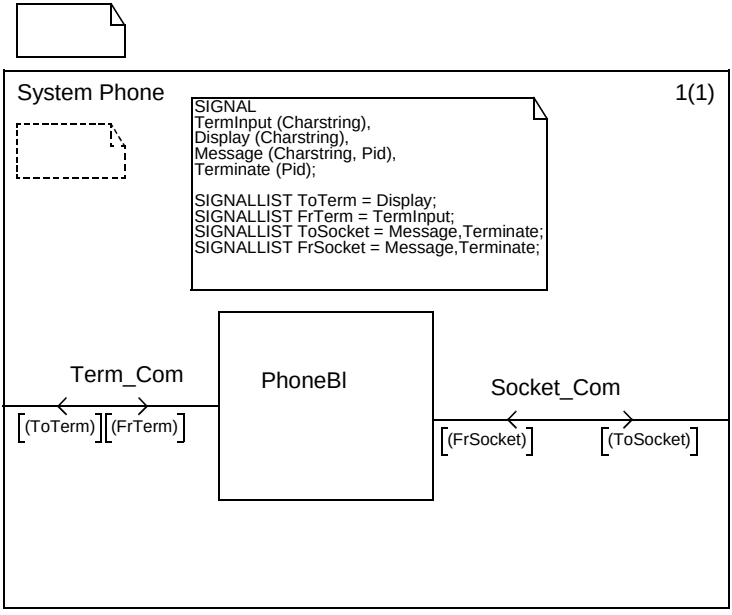


Figure 502: The system Phone

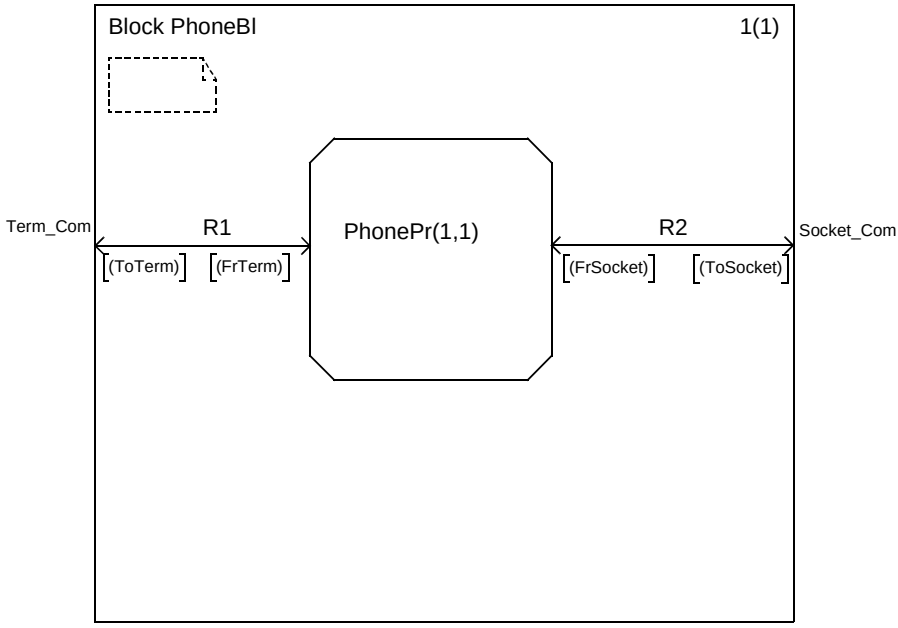


Figure 503: The block PhoneBl

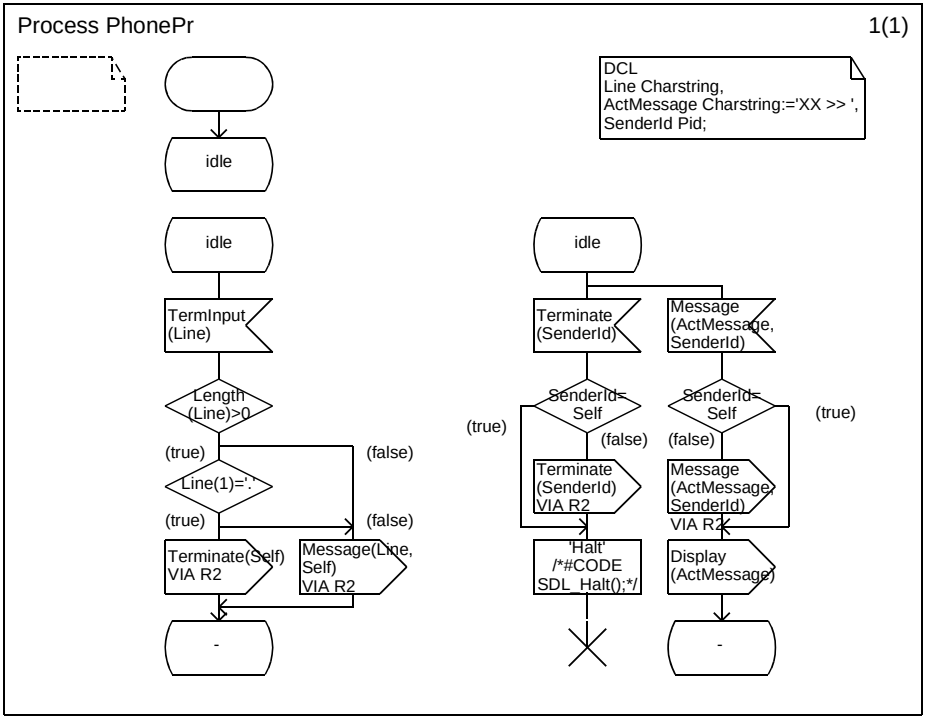


Figure 504: The process PhonePr

Appendix D: The Environment Functions

This section contains the environment functions included in the example. Note that this example is not updated to use the ASCII encoder.

```
/****+*******************************************************
00   sctEnv.c for SimplePhoneSys
*******************************************************/
#include "scttypes.h"
#include <stdio.h>

#include "phone.ifc"

#include <sys/types.h>
#include <sys/time.h>
#include <sys/socket.h>
#ifdef AIXV3CC
#include <sys/select.h>
#endif
#include <sys/un.h>

#include <unistd.h>
#define getdtablesize() ( (int) sysconf(_SC_OPEN_MAX) )

int Out_Socket, In_Socket;
struct sockaddr_un  Connection_Socket_Addr;
struct sockaddr_un  Connected_Socket_Addr;

#ifdef ULTRIXCC
#define PRINTF(s) \
  printf(s); \
  /* flush output to get a prompt */ \
  fflush( stdout)
#else
#define PRINTF(s) printf(s)
#endif

#ifdef XENV

/*#if !defined(XPMCOMM) && defined(XENV)*/
/*---+-------------------------------------------------------
     xGlobalNodeNumber  extern
-------------------------------------------------------------*/
#ifndef XNOPROTO
 int
xGlobalNodeNumber( void )
#else
 int
xGlobalNodeNumber()
#endif
{
  static int ProcId = -1;

  if (ProcId < 0)
    ProcId = getpid();
  return (ProcId);
}
/*#endif*/
```

```
/*---+-------------------------------------------------------
     xInitEnv  extern
-------------------------------------------------------------*/
#ifndef XNOPROTO
 void
xInitEnv( void )
#else
 void
xInitEnv()
#endif
{
  fd_set  readfds;
  int  addr_size;
  int  Connection_Socket;
  char  TmpStr[132];
  struct timeval  t;

  t.tv_sec = 60;
  t.tv_usec = 0;

  if ( (Connection_Socket = socket(PF_UNIX,SOCK_STREAM,0)) < 0
) {
    PRINTF("\nError: No Connection_Socket available!\n");
    SDL_Halt();
  }
  if ( (Out_Socket = socket(PF_UNIX,SOCK_STREAM,0)) < 0 ) {
    PRINTF("\nError: No Out_Socket available!\n");
    SDL_Halt();
  }

  sprintf(Connection_Socket_Addr.sun_path,
          "/tmp/Phone%d", xGlobalNodeNumber());
  Connection_Socket_Addr.sun_family = PF_UNIX;
  if ( 0 > bind(Connection_Socket, &Connection_Socket_Addr,
           strlen(Connection_Socket_Addr.sun_path)+2) ) {
    PRINTF("\nError: Bind did not succeed!\n");
    SDL_Halt();
  }
  listen(Connection_Socket, 3);

  sprintf(TmpStr, "\nMy Pid: %d\n", xGlobalNodeNumber());
  PRINTF(TmpStr);
  PRINTF("\nConnect me to: ");

  FD_ZERO(&readfds);
  FD_SET(1,&readfds);
  FD_SET(Connection_Socket,&readfds);
  if ( 0 < select(getdtablesize(), &readfds,
                  (fd_set*)0, (fd_set*)0, &t) ) {
    if ( FD_ISSET(1, &readfds) ) {
      (void)gets(TmpStr);
      sscanf(TmpStr, "%s", TmpStr);
      sprintf(Connected_Socket_Addr.sun_path, "/tmp/Phone%s",
              TmpStr);
      Connected_Socket_Addr.sun_family = PF_UNIX;
      if (connect(Out_Socket, (struct sockaddr
*)&Connected_Socket_Addr,
               strlen(Connected_Socket_Addr.sun_path)+2) < 0) {
        PRINTF("Error from connect\n");
        SDL_Halt();
      }
      FD_ZERO(&readfds);
      FD_SET(Connection_Socket,&readfds);
      if ( 0 < select(getdtablesize(), &readfds, (fd_set*)0,
                      (fd_set*)0, &t) ) {
        if ( FD_ISSET(Connection_Socket, &readfds) ) {
          addr_size =
```

```
strlen(Connection_Socket_Addr.sun_path)+2;
          In_Socket = accept(Connection_Socket,
&Connection_Socket_Addr,
                             &addr_size);
        }
      } else {
        PRINTF("\nError: Timed out\n");
        SDL_Halt();
      }
    }
    else if ( FD_ISSET(Connection_Socket, &readfds) ) {
      addr_size = strlen(Connection_Socket_Addr.sun_path)+2;
      In_Socket = accept(Connection_Socket,
&Connection_Socket_Addr,
                         &addr_size);
      FD_ZERO(&readfds);
      FD_SET(1,&readfds);
      if ( 0 < select(getdtablesize(), &readfds,
                      (fd_set*)0, (fd_set*)0, &t) ) {
        if ( FD_ISSET(1, &readfds) ) {
          (void)gets(TmpStr);
          sscanf(TmpStr, "%s", TmpStr);
          sprintf(Connected_Socket_Addr.sun_path,
"/tmp/Phone%s",
                  TmpStr);
          Connected_Socket_Addr.sun_family = PF_UNIX;
          if (connect(Out_Socket, (struct sockaddr
*)&Connected_Socket_Addr,
                  strlen(Connected_Socket_Addr.sun_path)+2) <
0) {
            PRINTF("Error from connect\n");
            SDL_Halt();
          }
        }
      } else {
        PRINTF("\nError: Timed out\n");
        SDL_Halt();
      }
    }
  } else {
    PRINTF("\nError: Timed out\n");
    SDL_Halt();
  }

  PRINTF("\n\n************ Welcome to SDT Phone System
************\n");
  PRINTF("\nphone -> ");
}

/*---+-----------------------------------------------------
     xCloseEnv  extern
-----------------------------------------------------------*/
#ifndef XNOPROTO
 void
xCloseEnv( void )
#else
 void
xCloseEnv()
#endif
{
  close(Out_Socket);
  close(In_Socket);
  unlink(Connected_Socket_Addr.sun_path);
  unlink(Connection_Socket_Addr.sun_path);
  PRINTF("\nClosing this session.\n");
}
```

```
/*---+-----------------------------------------------------------
     xInEnv  extern
-----------------------------------------------------------------*/
#ifndef XNOPROTO
 void
xInEnv( SDL_Time  Time_for_next_event )
#else
 void
xInEnv( Time_for_next_event )
  SDL_Time  Time_for_next_event;
#endif
{
  struct timeval t;
  fd_set         readfds;
  char          *Instr;
  int            NrOfReadChars;
  char           SignalName = '\0';
  xSignalNode    yOutputSignal;
  int            i = 0;
  char           chr = '\0';

  t.tv_sec = 0;
  t.tv_usec = 1000;
  FD_ZERO(&readfds);
#ifndef XMONITOR
  FD_SET(1,&readfds);
#endif
  FD_SET(In_Socket,&readfds);
  if ( select(getdtablesize(),&readfds,0,0,&t) > 0 ) {
#ifndef XMONITOR
    /*SDL-signal TermInput */
    if FD_ISSET(1, &readfds) {
      Instr = (char *)xAlloc(132);
      Instr[0]='L';
      Instr++;
      (void)gets(Instr);
      yOutputSignal = xGetSignal(TermInput, xNotDefPId, xEnv);
      xAss_SDL_Charstring(
        &((yPDP_TermInput)(OUTSIGNAL_DATA_PTR))->Param1, --
Instr,XASS);
      SDL_Output(yOutputSignal, (xIdNode *)NIL);
      xFree((void**)&Instr);
    }
#endif
    if FD_ISSET(In_Socket, &readfds) {
      Instr = (char *)xAlloc(151);
      do {
        read(In_Socket, &chr, 1);
        Instr[i++] = chr;
      } while ( chr!='\0' );
      sscanf(Instr, "%c", &SignalName);

      if ( SignalName == 'M' ) {
        /* SDL-signal Message */
        yOutputSignal = xGetSignal(Message, xNotDefPId, xEnv);
        sscanf(
          Instr+1,
          "%d %x%n",
          &(((yPDP_Message)(OUTSIGNAL_DATA_PTR))-
>Param2.GlobalNodeNr),
          &(((yPDP_Message)(OUTSIGNAL_DATA_PTR))-
>Param2.LocalPId),
          &NrOfReadChars);
        xAss_SDL_Charstring(
          &((yPDP_Message)(OUTSIGNAL_DATA_PTR))->Param1,
          (Instr+NrOfReadChars+2),XASS);
```

```
        SDL_Output(yOutputSignal, (xIdNode *)NIL);
      }
      else if ( SignalName == 'T' ) {
        /* SDL-signal Terminate */
        yOutputSignal = xGetSignal(Terminate, xNotDefPId,
xEnv);
        sscanf(
          Instr+1,
          "%d %x",
          &(((yPDP_Terminate)(OUTSIGNAL_DATA_PTR))-
>Param1.GlobalNodeNr),
          &(((yPDP_Terminate)(OUTSIGNAL_DATA_PTR))-
>Param1.LocalPId));
        SDL_Output(yOutputSignal, (xIdNode*)0);
      }
      xFree((void**)&Instr);
    }
  }
}

/*---+-----------------------------------------------------------
     xOutEnv  extern
-----------------------------------------------------------*/
#ifndef XNOPROTO
 void
xOutEnv( xSignalNode  *S )
#else
 void
xOutEnv( S )
  xSignalNode  *S;
#endif
{
  char   Outstr[150];

  /* SDL-signal Message */
  if ( (*S)->NameNode == Message ) {
    sprintf(Outstr,
            "M %d %x %.*s",
            ((yPDP_Message)((*S)))->Param2.GlobalNodeNr,
            ((yPDP_Message)((*S)))->Param2.LocalPId,
            strlen(((yPDP_Message)((*S)))->Param1),
            ((yPDP_Message)((*S)))->Param1);
    write(Out_Socket, Outstr, strlen(Outstr)+1);
    xReleaseSignal(S);
    return;
  }
  /* SDL-signal Terminate */
  if ( (*S)->NameNode == Terminate) {
    sprintf(Outstr,
            "T %d %x",
            ((yPDP_Terminate)((*S)))->Param1.GlobalNodeNr,
            ((yPDP_Terminate)((*S)))->Param1.LocalPId);
    write(Out_Socket, Outstr, strlen(Outstr)+1);
    xReleaseSignal(S);
    return;
  }
  /* SDL-signal Display */
  if ( (*S)->NameNode == Display ) {
    sprintf(Outstr, "\ndisplay ->%.*s",
           strlen(((yPDP_Display)((*S)))->Param1),
           ((yPDP_Display)((*S)))->Param1+1);
    PRINTF(Outstr);
    PRINTF("\nphone -> ");
    xReleaseSignal(S);
    return;
  }
}
#endif
```

Chapter 57 Building an Application

Chapter

58

ASN.1 Encoding and Decoding in the SDL Suite

This chapter is a reference manual for the ASN.1 encoding and decoding support in the SDL Suite. The support will help you to create applications and simulators that can both produce and decode protocol data units with bit patterns according to ASN.1 specifications and ASN.1 encoding rules. The SDL Suite supports BER (Basic Encoding Rules) and PER (Packed Encoding Rules).

In order for you to fully take advantage of this reference chapter, you should be familiar with the SDL Suite and the basics of ASN.1.

Note:

This chapter is not intended to describe the ASN.1 syntax or the general theory about ASN.1 encoding and decoding. This information can be found in the standard documents from ITU and from books about ASN.1.

Introduction

Supported Standards

The following standards are supported:

- ASN.1 according to ITU X.680, X.681, X.682 and X.683, 1997 version
- BER (Basic Encoding Rules) according to ITU X.690, 1997 version
- Basic PER (Packed Encoding Rules) according to ITU X.691, 1997 version, both aligned and unaligned variants.

Restrictions are listed in the Known Limitations chapter in the Release Guide.

Overview

The Abstract Syntax Notation One (ASN.1) is a notation language that is used for describing structured information that is intended to be transferred across some type of interface or communication medium. It is especially used for defining communication protocols.

SDL is suitable for specifying the semantic actions of a protocol, something that ASN.1 does not cover.

By using ASN.1 data types in the implementation of your application, you will optimize your development process. The following list displays some of the advantages of ASN.1:

- ASN.1 is a standardized, vendor-, platform- and language independent notation.
- A vast number of telecommunication protocols and services are defined using ASN.1. This means that pre-defined ASN.1 packages and modules are available and can be obtained from standardization organizations, RFCs, etc.
- When ASN.1 data types are transmitted over computer networks, their values must be represented in bit-patterns. Encoding and decoding rules determining the bit-patterns are already defined for ASN.1. The SDL Suite supports BER and PER encoding.

The SDL Suite support for ASN.1 consists of two major parts:

- An ASN.1 to SDL translator, ASN.1 Utilities, which allows you to use the ASN.1 types in your SDL systems.
- ASN.1 coders that will automatically encode your SDL values to bit-patterns and decode from bit-patterns to SDL values.

You can access the ASN.1 coders from SDL diagrams as well as from the C code.

Before you start the encoding and decoding steps, you must perform some preliminary steps. These are presented in chapter 8, *Tutorial: Using ASN.1 Data Types, in the Getting Started*.

Related Documents

Additional information about using ASN.1 and about encoding and decoding can be found in:

- chapter 8, *Tutorial: Using ASN.1 Data Types, in the Getting Started*
- chapter 13, *The ASN.1 Utilities*
- chapter 57, *Building an Application*
- chapter 65, *The Cmicro SDL to C Compiler*

Basic Concept

ASN.1 is suitable for specifying the information in a protocol and SDL is suitable for describing the semantic actions of a protocol. They are often used by standard bodies to specify protocol standards.

One of the main ideas behind ASN.1 is that you should be able to work with information in the protocol in the same way as you work with types and values in your implementation language. It should not be more difficult to create a bit-pattern in a protocol data unit then it is to assign a value to a variable. The encoder and decoder functions will automatically deal with all the details about bit-patterns and bit-layouts.

ASN.1 Utilities

The ASN.1 to SDL translator, *ASN.1 Utilities*, translates ASN.1 types and constants into SDL types and constants. An ASN.1 module generates an SDL package. This allows you to work with translated types and constants in the same way as any SDL type or constant. For instance, you can use them when you declare variables and when you declare signal parameters.

Information about ASN.1 to SDL translation and C representation of SDL data types is available in:

- chapter 13, *The ASN.1 Utilities*
- chapter 2, *Data Types, in the SDL Suite 6.2 Methodology Guidelines*

Encoding and Decoding

Encoding and decoding functionality consists of several parts.

First of all, it is the encoding and decoding algorithms themselves. You encode a variable or signal parameter into a bit-pattern by calling the encoder functions. You decode a bit-pattern into a variable or signal parameter by calling the decoder functions.

Another integral part of the coding functionality is the buffer handling. The decoder function takes the bit-pattern from the buffer and creates values for the application. The encoding function takes the internal value and encodes it into a sequence of bits in the buffer.

Error handling functionality is also present in the coding framework. Several types of errors can occur when encoding and decoding, such as buffer errors, incorrect values, decoding errors. Error handling functions allow to handle all possible erroneous situations.

Coding Access Interfaces

The coding functionality has got several access interfaces that allow you to access it in different ways.

It is possible to access encoding and decoding functions, call buffer functions and check error codes directly from the SDL diagram or from the C code in the environment files. For the C access interface, calls to encoder and decoder functions together with the appropriate buffer access procedures can be generated automatically if a few options in the user interface has been selected.

Basic SDL Interface and Extended SDL Interface

In SDL, you can use the basic interface or the extended interface. The basic interface is easy to use but gives you no possibility to control the buffer management. The extended interface gives you flexibility that the basic interface lacks.

The basic interface contains encode and decode functions which operates at SDL octet strings. The extended interface adds the possibility to access the buffer interface from within SDL and operates at a new type call CoderBuf. Furthermore, the extended interface contains a new type called CustomCoderBuf with which you could implement your own mapping between SDL and the encode and decode functions.

Within an SDL system, it is only possible to use one type of SDL interface. Either the SDL Basic Interface, the SDL Extended Interface with CoderBuf or the SDL Extended Interface with CustomCoderBuf.

C Code Interface

The coder library contains encoding and decoding functions and also several help functions for buffer management, error management and printing information. If you choose to call encoding and decoding from outside the environment files, note that:

- You need a file named `<ASN.1 module>.ifc` with necessary C declarations of types, literals and operators. The SDL to C Compiler

generates this file if you have added the ASN.1 module to your SDL system with a use package clause.

- You need generated C files and static kernel C files. These files will be present if you have generated an SDL system with your ASN.1 module.

General information about using .ifc files and writing C code with SDL is available in:

- chapter 57, *Building an Application*
- chapter 65, *The Cmicro SDL to C Compiler*

Solution

This section describes the structure and design of the encoding and decoding support in the SDL Suite. It is intended to give you an understanding of the solution.

Functionality

The main goal is to help you to develop executable files, applications or simulators, that can:

- produce BER/PER bit patterns
- consume and decode BER/PER bit patterns
- be configured to work with user specific environment (memory management, buffer management and error handling)

ASN.1 coder framework contains the following functional modules:

- Encoding and decoding functionality (see “Encoding and Decoding Functionality” on page 2838)
- Buffer Management System (BMS) (see “Buffer Management System” on page 2850)
- Memory Management System (MMS) (see “Memory Management System” on page 2866)
- Error Management System (EMS) (see “Error Management System” on page 2870)
- Debug print opportunities (see “Printing Opportunities” on page 2885)

All these modules will be described in more details in this chapter.

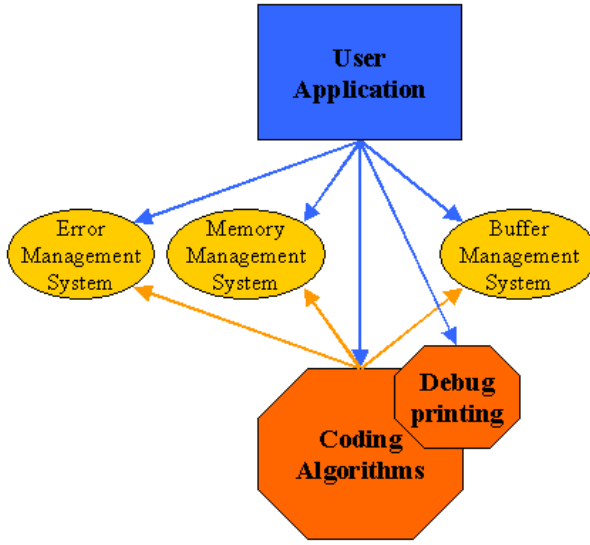


Figure 505: Relationship between functional modules of the encoding/decoding solution

Functionality Access Interfaces

When using the coder functionality, you can choose between different access interfaces. They all share the same implementation.

- **Basic SDL interface**

  SDL operators that put bytes in octet strings and read bytes from octet strings. Buffer access functions are not visible in this SDL interface. For more information see "Encoding/decoding the SDL Basic Interface" on page 2842.

- **Extended SDL interface**

  SDL `encode` and `decode` operators that put bytes in the types CoderBuf and CustomCoderBuf. Parts of the buffer management is

visible in the interface. For more information see "Encoding/decoding the SDL Extended Interface" on page 2844.

- **C code interface**

  This interface is used when calling the coders from C code, for example in the environment files. For more information see "C Encoding and Decoding Interface" on page 2838.

The SDL and C interfaces are not identical. They are adapted to the properties of the languages.

It is preferred to access the coders from SDL when the encoded bit-pattern must be further processed in the SDL system. One interesting example is when the bit pattern is sent in a signal to another process.

It is preferred to access the coders in C-code in the environment files when the SDL-system does not process the bit pattern internally, for example when sending it to a protocol in the environment.

Encoding and Decoding Functionality

Encoding and Decoding Functions

An encoding function encodes an SDL variable or signal parameter into a bit-pattern according to BER or PER. A decoding function decodes from a PER or BER bit-pattern to an SDL variable or signal parameter. You can access the encoding and decoding functions either from the C code (see "C Encoding and Decoding Interface" on page 2838) or from SDL diagrams (see "SDL Encoding and Decoding Interfaces" on page 2842).

C Encoding and Decoding Interface

The most common example of implementing coders in C is in the environment file. The SDL to C Compiler automatically generates an environment file with all needed calls. The coders can be accessed from any function or module implemented in C code.

There is one set of functions for BER and one set of functions for PER. The functions are accessed by using macros `BER_ENCODE` and `BER_DECODE`, respectively `PER_ENCODE` and `PER_DECODE`.

There is also a set of common functions which choose encoding rules dynamically according to the parameter written to the buffer by function `BufSetRule(buffer, rule)`, see "Buffer Management Functions" on page 2853. The functions are accessed by using macros `ASN1_ENCODE` and `ASN1_DECODE`.

Example 471 Dynamic encoding rules configuration ----------

```
   BufInitWriteMode(buf);
   BufSetRule(buf, er_PER | er_Aligned);
/* PER Aligned will be applied */

   ASN1_ENCODE( buf, (tASN1TypeInfo *)&yASN1_MyType1,
                (void *)&((yPDP_Sig1)(*S))->Param1));
   BufSetRule(buf, er_BER | er_Definite);
/* BER Definite will be applied */

   ASN1_ENCODE( buf, (tASN1TypeInfo *)&yASN1_MyType2,
                (void *)&((yPDP_Sig2)(*S))->Param1));
   BufCloseWriteMode(buf);
```

All the encode and decode macros, `ASN1_ENCODE`, `ASN1_DECODE`, `BER_ENCODE`, `BER_DECODE`, `PER_ENCODE` and `PER_DECODE` have three parameters:

- **First parameter**:

  The first parameter is a reference to a buffer of type `tBuffer` (see "Types and definitions" on page 2851). During encoding the resulting bit pattern will be stored in a memory section associated with the buffer. The buffer must be in the write mode when the `encode` function is called, because it writes bytes to the buffer. During decoding a bit pattern will be read from a memory section associated with the buffer. The buffer must be in read mode when the `decode` function is called, because it reads bytes from the buffer. For more information about buffer handling see "Buffer Management System" on page 2850.

- **Second parameter**:

  Pointer to ASN.1 type information (see "ASN.1 Type Information Generated by ASN.1 Utilities" on page 2841). All information that is needed about the type in the ASN.1 specification is represented in this type information. The reference to type information is of type `tASN1TypeInfo*`. The type information structure is a global variable with a name `yASN1_<type name>`. The reference to the type information structure is then `(tASN1TypeInfo*)&yASN1_<type name>`.

- **Third parameter**:

  Memory pointer to variable or signal parameter to encode or decode. The memory pointer is of type `void*` .

The buffer macro returns an error code which is a literal from an enumerated type (see "Error codes" on page 2873). The return value indicates whether the encoding or decoding procedure was successful or not. If this value is `ec_SUCCESS`, then procedure was successful.

Example 472: BER encoding of signal parameter ––––––––––––––––

In ASN1:

```
Message ::= SEQUENCE {
                        id     INTEGER,
                        info   INFOTYPE
                      }
```

In SDL:

```
SIGNAL Sig2(Message);
```

In xOutEnv:

```
BufInitWriteMode(buf);
BER_ENCODE( buf,
           (tASN1TypeInfo *)&yASN1_Message,
            (void *)&((yPDP_Sig2)(*S))->Param1));
BufCloseWriteMode(buf);
```

Example 473: PER encoding of signal parameters

In xOutEnv:

```
BufInitWriteMode(buf);
PER_ENCODE( buf,
            (tASN1TypeInfo *)&yASN1_Message,
            (void *)&((yPDP_Sig2)(*S))->Param1));
BufCloseWriteMode(buf);
```

Example 474: BER decoding of signal parameters

In xInEnv:

```
BufInitReadMode(buf);
BER_DECODE( buf,
           (tASN1TypeInfo *)&yASN1_Message,
            (void *)&((yPDP_Sig2)(S))->Param1));
BufCloseReadMode(buf);
```

Example 475: PER decoding of signal parameters

In xInEnv:

```
BufInitReadMode(buf);
PER_DECODE( buf,
           (tASN1TypeInfo *)&yASN1_Message,
            (void *)&((yPDP_Sig2)(S))->Param1));
BufCloseReadMode(buf);
```

ASN.1 Type Information Generated by ASN.1 Utilities

The encoder and decoder functions need information about the types from the ASN.1 specifications. This information is stored in type information structures, called type information nodes or simply type nodes. Almost all information about the types that you can find in the ASN.1 specifications is stored in the type information nodes.

The internal structure of the type information nodes is important in an implementation of encoding and decoding functions, but not from the application that calls the functions. There, it is only important to find a reference to a node. The internal details of the type nodes are not described in this section.

The type information nodes are represented in C code as a set of global read-only variables. The information in a node cannot be changed during execution. The nodes are linked to each other by using memory references. The type nodes can be declared as `const` variables by defining a macro, which means that in some application they can be moved to program memory or read only memory. For more information, see "Compilation switches" on page 2891.

The name of a node is `yASN1_<type name>`. A reference to a type node is simply a memory pointer to the type node cast to a basic type node declaration.

Example 476: Reference to a type node ––––––––––––––––––––––

If the type name is `Pdu1` in the ASN.1 specification, the reference to the type node is:

```
(tASN1TypeInfo*)&yASN1_Pdu1.
```

––

The nodes reflect the information in ASN.1 specifications and must be generated by ASN.1 Utilities. ASN.1 Utilities parses and analyzes the ASN.1 information and generates C code containing type nodes. One .c-file and one .h file per ASN.1 module is generated and the files are named `<module name>_asn1coder.c` and `<module name>_asn1coder.h`.

SDL Type and Operator Information Generated by the SDL to C Compiler

The encoder and decoder functions also need information generated by the SDL to C compiler, for example declarations of operators and literals for manipulating the translated ASN.1 types. This information is also stored in the type nodes.

ASN.1 utilities generates one SDL package for each ASN.1 module. The SDL to C Compiler generates one `.c` and `.h` file for each package, with type and operator information, together with one `.ifc` file for an external view of the types and operators. For more detailed information about the C code generation see "System Interface Header File" on page 2777 in chapter 57, *Building an Application*.

The ASN.1 type information files includes the `<package>.ifc` files. All relationships are resolved during compile and link time and the information is read-only at execution time.

SDL Encoding and Decoding Interfaces

In the SDL interface, the ASN.1 type information pointer is not seen. It is implicit and derived from the type of the memory pointer.

There are two types of coder interfaces available from within an SDL system: SDL basic interface (see "Encoding/decoding the SDL Basic Interface" on page 2842) and SDL extended interface (see "Encoding/decoding the SDL Extended Interface" on page 2844). In the SDL basic interface, the buffer reference is replaced with an octet_ string and the encoded bit-pattern is put directly in a variable of octet_string type. In the SDL extended interface, the buffer reference is CoderBuf or CustomCoderBuf.

Encoding/decoding the SDL Basic Interface

You can encode and decode directly from the SDL diagram by using the operators `encode` and `decode`.

The syntax of the `encode` function is:

```
result:= encode(encoded_octet_string, ASN1_variable)
```

The two parameters of the `encode` function are defined as:

- `encoded_octet_string` is an `Octet_string` that contains the encoded value of the ASN1_variable if the encoding was successful.

- `ASN1_variable` is the ASN.1 variable to be encoded.

The function returns an integer, `result`. If the value is equal to `ec_SUCCESS` the encoding was successful. For more information about return status see "SDL error interface" on page 2873.

The encoded value in the octet string could then be used in a signal sent to an other part of the system or to the environment.

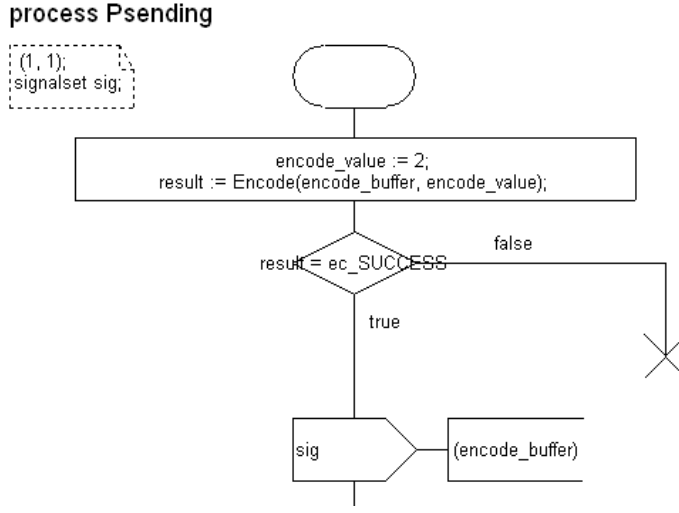


Figure 506 Example of a encode call in an SDL diagram

The syntax of the `decode` function is:

```
result:= decode(octet_string_to_decode,
ASN1_variable)
```

The two parameters of the `decode` function are defined as:

- `octet_string_to_decode` is an `Octet_string` that contains the encoded value to decode.
- `ASN1_variable` is the ASN.1 variable that will contain the result after the decoding was performed.

The function returns an integer, `result`. If the value is equal to `ec_SUCCESS` the encoding was successful. For more information about return status see "SDL error interface" on page 2873.

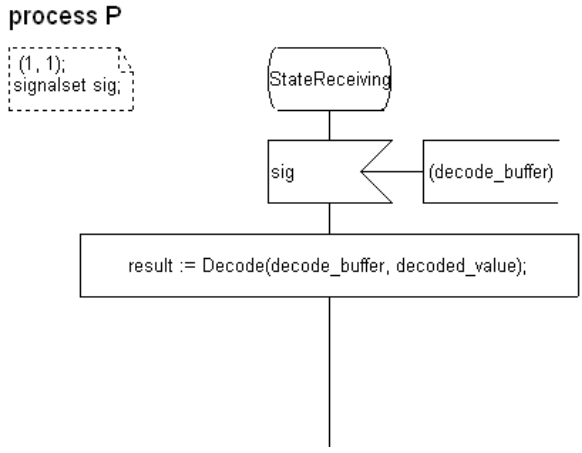


Figure 507 Example of a encode call (SDL Basic Interface)

Encoding/decoding the SDL Extended Interface

As with the Basic SDL Interface you can encode and decode directly from the SDL diagram by using the operators `encode` and `decode`.

The syntax of the `encode` function is:

```
result:= encode(CoderBuf, ASN1_variable)
```

The two parameters of the `encode` function are defined as:

- `CoderBuf` is a type defined in the file `CoderBuf.sdl` . It is a mapping of the buffer management C-interface (see "C interface to buffer management system" on page 2851). For more information about operations at CoderBuf see "SDL CoderBuf interface" on page 2862.
- `ASN1_variable` is the ASN.1 variable to be encoded.

The function returns an integer, `result`. If the value is equal to `ec_SUCCESS` the encoding was successful. For more information about return status see "SDL error interface" on page 2873.

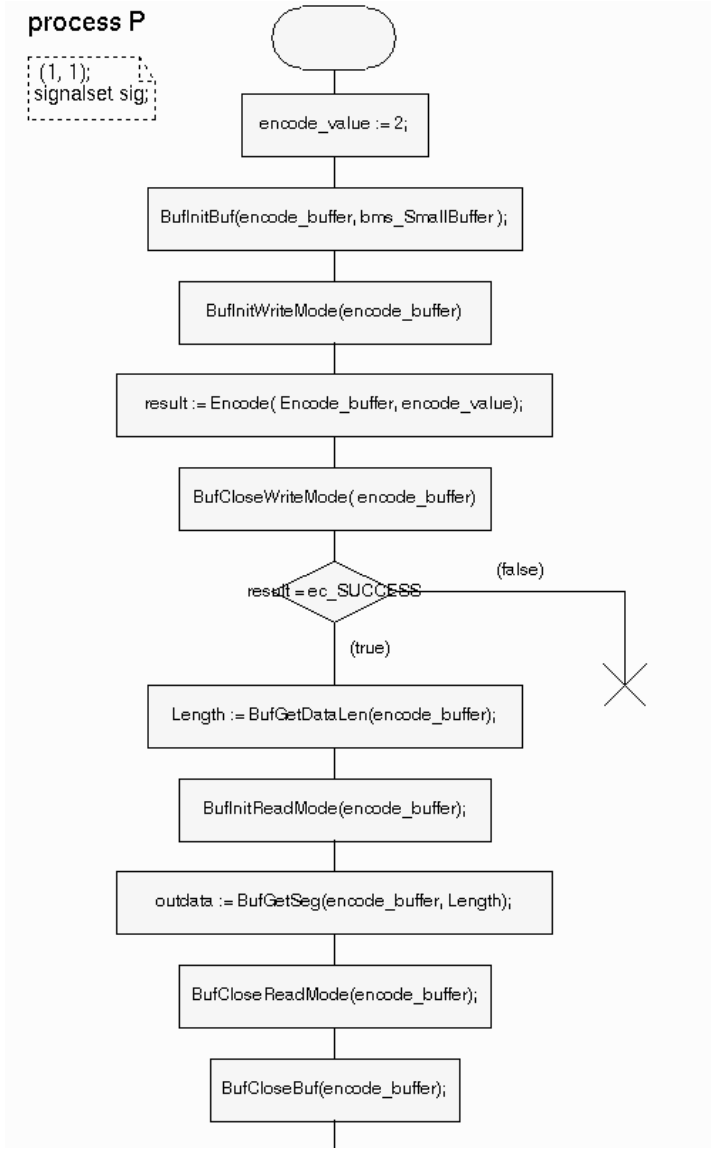


Figure 508: Example of en encode call (SDL Extended interface)

The syntax of the `decode` function is:

```
result:= decode(CoderBuf, ASN1_variable)
```

The two parameters of the `decode` function are defined as:

- `CoderBuf` is a type defined in the file `CoderBuf.sdl`. It is a mapping of the buffer management C-interface (see "C interface to buffer management system" on page 2851). For more information about operations at CoderBuf see "SDL CoderBuf interface" on page 2862.
- `ASN1_variable` is the ASN.1 variable that will contain the result after the decoding was performed.

The function returns an integer, `result`. If the value is equal to `ec_SUCCESS` the encoding was successful. For more information about return status see "SDL error interface" on page 2873.

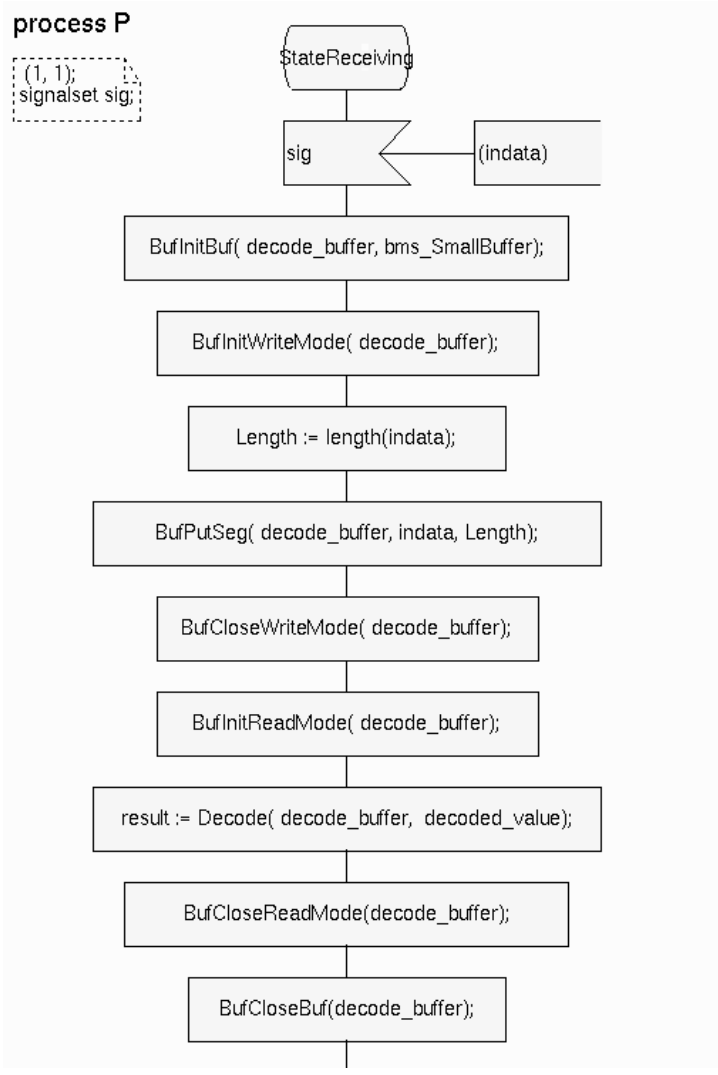


Figure 509: Example of decode call (SDL Extended Interface)

How to Include the Extended SDL Interface in Your System

Add the package in the Organizer. In the directory `<installation_directory>/sdt/include/ADT/` there are two versions of the BufferInterface package. One for C, `CoderBuf.sun`, and one for C++, `CoderBufCPP.sun` . When the package is included in the Organizer, add a use statement of the BufferInterface in your system.

> **Note: Types in the BufferInterface package**
>
> In the BufferInterface package BasicCTypes and CPointer or the corresponding C++ versions of those files are included. This must be considered when using `cpp2sdl` together with your system so that those files are included only once.

The Extended SDL Interface in the Simulator

You can display the value of a CoderBuf and change it when running the Simulator by using the commands examine-variable and assign-variable. If you have written your own buffer you can add information in the provided files to be able to use them in the Simulator, see below.

Examine CoderBuf

Command: ex-va encode_buffer

encode_buffer (CoderBuf) = bms_UserBuffer(NM)3'020102'H

bms_UserBuffer is the buffer type, (NM) shows the current mode of the coderbuffer(NoMode, ReadMode or WriteMode), 3 is the length of the coder buffer value in bytes and '020102'H is the value of the coder buffer in octet string form

Limitations:

Examining the value when the coder buffer is in Write Mode is not possible.

If the user has defined a new buffer type other than Small Buffer and User buffer the buffer type will not be displayed.

Example:

encode_buffer (CoderBuf) = BufferType(NM)3'020102'H

To display the right buffer type the user has to do some modifications in the write function (yWri_CoderBuf) in the file CoderBuf.sdl.

Only the first 100 bytes of a coder buffer value are displayed during simulation.

Assign new values to a coder buffer:

Command: ass-va encode_buffer 2 '0101'H ReadMode(RM)

Value assigned:

encode_buffer (CoderBuf) = bms_UserBuffer(RM)2'0101'H

When you assign value to a coder buffer:

Command: ass-va <length> <value> <Mode>

Length is the length of the value in bytes. Value is the new value (octet string). Mode, the user sets the mode of the coder buffer (Read, Write and No).

In order for the assignment to be successful the user has to give the two first parameters; otherwise the command will not be executed. An error message will be displayed to the user.

If the length of the value passed as parameter is less than the length passed as parameter, extra zeros (0) are added to the octet string to match the length passed by the user. If the length of the value passed as parameter is greater than the length passed as parameter, then the octet string to be assigned will be cut to match the length.

If the user does not choose the buffer mode, the buffer mode will be set to No Mode (default).

Limitations:

If the user has defined a new buffer type other than Small Buffer and User buffer, the user has to do some modifications in the read function (yRead_CoderBuf) in CoderBuf.sdl in order to initiate the new coder buffer to be assigned to the old coder buffer with right coder buffer type.

The maximum size of the coder buffer is assumed to be 128 Mega bytes.

Buffer Management System

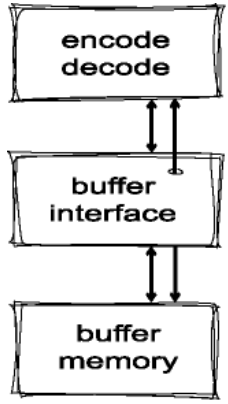


Figure 510: Relationship between coders and buffer management

The buffer interface is a set of functions that operates with data. Buffer management stores data in the buffer memory. Data is presented as `byte`, `bit` or `character segment`. Available basic operations are `get`, `put`, `peek` and `skip`. The buffer interface also opens a possibility for direct access to buffer memory from BMS environment (from encode and decode). According to this there is direct access to the buffer memory which presented as one continuos piece.

The buffer management functions have an open design, where it is possible to choose between different buffer implementations with different characteristics. This choice is available in runtime mode and also within compilation by switches defined in "Buffers configuration" on page 2896. This is possible due to an open and specified buffer interface that all encode and decode functions use and that your SDL specifications and C functions should use as well. The interface is specified as a set of C macros. See "C interface to buffer management system" on page 2851 for more information. The buffer management implements all the macros in the interface.

The buffer management is accessed from an SDL system by using operators in the buffer management Abstract Data Type. The operators are mapped to the open buffer management interface.

The only buffer management implementation at present in the SDL Suite is the small buffer (see "Small Buffer Implementation" on page 2862).

C interface to buffer management system

This chapter describes the open buffer interface. This is the C access interface to the buffer management.

Types and definitions

The buffer access type is `tBuffer`. `tBuffer` is defined as a pointer to a structure `tCoder`.

Example 477: Buffer as a reference ––––––––––––––––––––––––––

```
tBuffer buf = NULL;
BufInitBuf(buf, bms_SmallBuffer);
```

`tCoder` is used as access type only in tiny mode (see Example 487 on page 2896).

BMS introduces several specific types which are described below.

tBMSLength

BMS length type is defined as `unsigned long`.

tBMSUserMemory

This type is structure describing information about memory provided by the user. It contains three fields:

- `MemSize` of type `tBMSLength` - size of the provided memory
- `DataLength` of type `tBMSLength` - size of the data written to the provided user memory
- `MemPtr` of type `void *` - pointer to the provided memory

This type is used as parameter type in the buffer interface functions `BaseBufInitBufWithMemory` and `BaseBufCloseBufToMemory`.

tBMSBufType

This type is an enumeration describing all following possible types of buffers. Two types of buffers are available:

- `bms_SmallBuffer` - predefined buffer type (see "Small Buffer Implementation" on page 2862)
- `bms_UserBuffer` - type denoting user defined buffer (see "User defined Buffer Handling" on page 2864)

This type is used for specifying the type of buffer which will be used.

tERRule

This type is an enumeration describing possible kinds of encoding rules. It contains basic rules (BER, PER, etc.) as well as their different variations:

- `er_BER`
- `er_Definite`
- `er_Indefinite`
- `er_PER`
- `er_Aligned`
- `er_Unaligned`
- `er_NoEndPad` - PER without end padding
- `er_UER` - User defined encoding rules
- `er_NoRule`

The enumeration values are regarded as bit masks for different kinds of encoding rules, for example, PER Aligned can be specified with `er_PER | er_Aligned`; expression `( SpecifiedRule & ( er_PER | er_Aligned ) )` will check if the specified type of encoding rules `SpecifiedRule` is PER Aligned.

tBMSMode

This type is an enumeration describing possible modes for the buffer:

- `bms_ReadMode` - reading mode, writing is allowed for appending. See `BufSetAppendBufFunc( buffer, func )` for more information.
- `bms_WriteMode` - writing mode, reading is not allowed
- `bms_NoMode`

Buffer Management Functions

The buffer management functions put bytes and bits into a buffer which you can use to send over a protocol. The buffer management also provide several interfaces to operate with buffer memory and buffer handling.

The internal implementation of the buffer is not shown in the buffer interface, thus all manipulation of the buffer should be done using the macros. A buffer must be created before use. It must be opened before a read/write session and closed after a session. A buffer can be in either a read mode or a write mode, but not in both modes at the same time.

The open buffer interface macros are described below (parameter `buffer` in all functions is of type `tBuffer`, see "Types and definitions" on page 2851):

- `BufInitBuf( buffer, buffer_type )`

  This function is used to initialize the buffer management. Memory used by the buffer is allocated by `CUCF_ALLOC` function, so memory management used in the buffer handling can be configured by means of MMS configuration opportunities (see "User defined Memory Handling" on page 2868). The second parameter is the type of the buffer (see "tBMSBufType" on page 2851). The function returns error status (see "Error codes" on page 2873).

- `BufCloseBuf( buffer )`

  Closes a buffer and if the buffer has not been initialized by the user memory, releases buffer memory by using `CUCF_FREE`. The function returns error status (see "Error codes" on page 2873).

- `BufInitBufWithMemory( buffer, buffer_type, user_memory)`

  This function initializes the buffer with the user memory provided in the third parameter (see "tBMSUserMemory" on page 2851) without allocating any memory. If user memory contains any encoded data, the buffer can be used further by the decode function. The second parameter is the type of the buffer (see "tBMSBufType" on page 2851). The function returns error status (see "Error codes" on page 2873).

- `BufCloseBufToMemory( buffer, user_memory )`

This function closes the buffer. It does not release any memory, but returns it in the `user_memory` parameter of type "tBMSUserMemory" on page 2851 passed by reference. The function returns error status (see "Error codes" on page 2873).

- `BufGetMemory( buffer, length )`

  Returns pointer to the buffer memory. The second parameter is used for returning. It is pointer to the length of memory.

- `BufAppendMemory( buffer, append_length )`

  Used for the appending of the buffer memory with required `append_length`. Returns pointer to the beginning of the memory (root).

- `BufGetDataCurr( buffer, bit_position )`

  Returns current data pointer in the buffer handling. The second parameter is used for returning. It is a pointer to a char with current bit position in the buffer handling. This function is used by the decoder.

- `BufSetDataCurr( buffer, length, bit_position )`

  Used to increment current data pointer by `length` in the buffer handling and also sets current bit position to `bit_position` there. This function is used by the decoder.

- `BufGetDataEnd ( buffer, bit_position )`

  Returns end data pointer in the buffer handling. The second parameter is used for returning. It is a pointer to a char with the end bit position in the buffer handling. This function is used by the encoder and decoder.

- `BufSetDataEnd ( buffer, length, bit_position )`

  Uses to increment end data pointer by `length` in the buffer handling and also sets end bit position to `bit_position` there. This function is used by encoder.

- `BufCopyBuf( buffer_dst, buffer_src )`

  This function copies the source buffer `buffer_src` to target buffer `buffer_dst`. The function returns error status (see "Error codes" on page 2873).

- `BufInitReadMode( buffer )`

Prepares the buffer for read operations (read mode, see "tBMSMode" on page 2852) and puts the current pointer to the start of the buffer data. The buffer content is not changed or released. The function returns error status (see "Error codes" on page 2873).

- `BufCloseReadMode( buffer )`

  Closes read mode for the buffer. The buffer content is not changed or released. The function returns error status (see "Error codes" on page 2873).

- `BufCloseDeleteReadMode( buffer )`

  Closes read mode for the buffer and logically (without memory releasing) removes read bytes from buffer. Bytes that are not read are still in the buffer. The function returns error status (see "Error codes" on page 2873).

- `BufInitWriteMode( buffer )`

  Prepares the buffer for write operations (write mode, see "tBMSMode" on page 2852) and sets the current pointer to the start of the buffer. Previous information in buffer is logically lost (the bytes are still there but they are considered to be empty). The function returns error status (see "Error codes" on page 2873).

- `BufCloseWriteMode( buffer )`

  Closes write mode for the buffer. The buffer content is not changed or released. The function returns error status (see "Error codes" on page 2873).

- `BufGetBufType( buffer )`

  Returns the type of the buffer, see "tBMSBufType" on page 2851. It can be useful when the system supports more than one buffer interface and the type of the buffer is not obvious and can change dynamically.

- `BufGetDataLen( buffer )`

  Returns the byte length (see "tBMSLength" on page 2851) of all data that is physically present in the buffer.

- `BufGetDataBitLen( buffer )`

  Returns the bit length (see "tBMSLength" on page 2851) of all data that is physically present in the buffer.

- `BufGetReadDataLen( buffer )`

  Returns the byte length (see "tBMSLength" on page 2851) of the information which has been read from the buffer.

- `BufGetReadDataBitLen( buffer )`

  Returns bit length (see "tBMSLength" on page 2851) of the information which has been read from the buffer.

- `BufGetValueLen( buffer )`

  Returns byte length (see "tBMSLength" on page 2851) of the last value encoded to the buffer or decoded from the buffer.

- `BufGetValueBitLen( buffer )`

  Returns bit length (see "tBMSLength" on page 2851) of the last value encoded to the buffer or decoded from the buffer.

- `BufGetMode( buffer )`

  Returns buffer mode, see "tBMSMode" on page 2852.

- `BufInReadMode( buffer )`

  This is a boolean function returning `true` if the buffer is in the read mode (see "tBMSMode" on page 2852).

- `BufInWriteMode( buffer )`

  This is a boolean function returning `true` if the buffer is in the write mode (see "tBMSMode" on page 2852).

- `BufInNoMode( buffer )`

  This is a boolean function returning `true` if the mode of the buffer is not defined.

- `BufSetRule( buffer, rule )`

  This function sets the encoding rules to the buffer. Each setting overrides the previous. They are checked later in the encode and decode functions. The second parameter is combination of encoding rule masks of type `tERRule` (see "tERRule" on page 2852), it is automatically cast to `unsigned long`, which is the type of the second parameter. For example, PER without end padding encoding rule is set with an expression `BufSetRule(er_PER | er_NoEndPad)`. If not a complete set of rules is specified then `BufSetRule` will use default settings, for example `BufSetRule(er_PER)` is used (with-

out bit masks `er_Aligned`, `er_Unaligned` and `er_NoEndPad`), then default PER variant will be applied which is configured by compilation switches, see "Compilation switches" on page 2891. The BufGetRule will return the encoding rules including those that have been applied depending on default settings.

- `BufGetRule( buffer )`

  This function returns value of type `unsigned long` which can be checked by binary "AND" operation with bit masks of type `ERRule` (see "tERRule" on page 2852), for example, `(SpecifiedRule & er_PER)` returns true if specified rule is PER.

Caution! Checking specified encoding rules

```
unsigned long SpecifiedRule = BufGetRule( buffer );
```

`( SpecifiedRule & er_PER & er_Aligned )` is not a correct check for PER Aligned.

`( SpecifiedRule & ( er_PER | er_Aligned ) )` should be used instead.

- `BufGetUserData( buffer )`

  This function returns a reference to the user data defined by `tUser-Data` type (see "User Data" on page 2884).

- `BufSetErrInitFunc( buffer, func )`

  This function sets a reference to the user error handling function. The interface is available only if `CODER_REMOVE_PATH` compile switch is absent (see "Error handling configuration" on page 2898).

- `BufGetErrInitFunc( buffer )`

  This function returns a reference to the user error handling function. `NULL` is returned if the function has not been set. The interface is available only if `CODER_REMOVE_PATH` compile switch is absent (see "Error handling configuration" on page 2898).

- `BufGetErrorPath( buffer )`

  This function returns a reference to the `tErrorPath` structure that contains a path of fields from the root type up to the field where an error has occurred. The C-representation of the `tErrorPath` is:

  ```
  typedef struct
  ```

```
{
  void*          Fields[CODER_PATH_DEEP];
  unsigned long  NumOfFields;
  ...
} tErrorPath;
```

where `CODER_PATH_DEEP` is the maximum number of nested types, see "Error handling configuration" on page 2898. It is used within the user error handling function. The interface is available only if `CODER_REMOVE_PATH` compile switch is absent (see "Error handling configuration" on page 2898). The first element in the `Fields` array is a reference to the root Type Info - `tASN1TypeInfo*`. The next element might be one of the following:

`tASN1Component*` - for the SEQUENCE/SET component

`tASN1Alternative*` - for the CHOICE alternative

`tASN1Object*` - for an open object

`long*` - for the SEQUENCE OF/SET OF item index

- `BufGetMainVal( buffer )`

  This function returns a reference to the encode/decode value and it is used in the user error handling to assign the default contents to the value if an error has occurred. The interface is available only if `CODER_REMOVE_PATH` compile switch is absent(see "Error handling configuration" on page 2898).

- `BufSetErrorCode( buffer, error_code )`

  This function sets an error code (see "Error codes" on page 2873) to the buffer.

- `BufGetErrorCode( buffer )`

  This function returns an error code (see "Error codes" on page 2873).

- `BufGetByte( buffer )`

  Reads one byte from the buffer and returns it in a variable of type `unsigned char` after logically removing it from the buffer. The byte that has been returned will not be read from the buffer once again by another reading procedure. This function returns `unsigned char`.

- `BufPeekByte( buffer )`

Takes one byte from the buffer and returns it in a variable of type `unsigned char` without removing it from the buffer. The byte that has been returned can be peeked from the buffer any number of times. This function returns `unsigned char`.

- `BufPutByte( buffer, byte )`

  Puts one byte from the second parameter of type `unsigned char` to the buffer.

- `BufGetSeg( buffer, length )`

  Reads a segment of length (see "tBMSLength" on page 2851) bytes from a buffer and returns a segment pointer that points at the start of the retrieved segment. Note that this pointer should not be freed, because it is pointing into the memory segment of the buffer. The segment that has been returned will not be read from the buffer once again by another reading procedure. This function returns `unsigned char*`.

- `BufSkipSeg( buffer, length )`

  Skips `length` (see "tBMSLength" on page 2851) bytes from the buffer.

- `BufPeekSeg( buffer, length )`

  Takes `length` (see "tBMSLength" on page 2851) bytes from the buffer and returns a segment pointer from buffer without moving current pointer in a variable of type `unsigned char*`. The segment that has been returned can be peeked from the buffer any number of times. Note that this pointer should not be freed, because it is pointing into the memory segment of the buffer.

- `BufPutSeg( buffer, segment, length )`

  Puts a whole segment of `length` (see "tBMSLength" on page 2851) bytes to the buffer. The second parameter `segment` is of type `unsigned char*` and is a pointer to the segment start.

- `BufGetBit( buffer )`

  Reads one bit from the buffer and returns it in a variable of type `unsigned char` after logically removing it from the buffer. The bit that has been returned will not be read from the buffer once again by another reading procedure. This function returns `unsigned char`, where returned bit is the right-most bit in the `unsigned char` value.

- `BufPutBit( buffer, bit )`

  Puts one bit which is the right-most bit of function parameter of type `unsigned char` to the buffer.

- `BufGetBits( buffer, number )`

  Reads `number` bits from the buffer and returns them in `number` right-most bits in a variable of type `unsigned long` after logically removing them from the buffer. The bits that have been returned will not be read from the buffer once again by another reading procedure. The second parameter `number` is of type `unsigned char`. Bits are returned in a variable of type `unsigned long`, so `number` should be less or equal to `sizeof(unsigned long)*8`.

- `BufPutBits( buffer, bits, number )`

  Puts `number` right-most bits from a variable of type `unsigned long` to the buffer. Bits are passed in a variable of type `unsigned long`, so `number` should be `<= sizeof(unsigned long)*8`.

- `BufAlign( buffer )`

  This procedure skips all the bits from the buffer till the end of the byte.

- `BufSetAppendBufFunc( buffer, func )`

  Sets a reference to an append function. The append function is called when there are not enough bytes left in the buffer for a get operation and more bytes must be put into the buffer. Default is no append function.

Example 478: Sending a buffer ----------

```
unsigned char * data;
unsigned int    datalen;

BufInitReadMode(buf);
datalen = BufGetDataLen(buf);
data = BufGetSeg(buf,datalen);
send_protocol(sa,data,datalen);
BufCloseReadMode(buf);
```

Example 479: Receiving from protocol and putting in a buffer ------

```
unsigned char data[MAXSIZE];
unsigned int datalen;
```

```
BufInitWriteMode(buf);
datalen=1;
while(datalen>0) {
   datalen = receive_protocol(sa,data,MAXSIZE,0);
   if (datalen>0)
      BufPutSeg(buf,data,datalen);
}
BufCloseWriteMode(buf);
```

You can write an append function that will be called when there are not enough bytes left in the buffer for a read operation. An example of when this can occur is during the execution of a decode function. The calling function calculates how many bytes that must be added to the buffer. The append function can choose to add more bytes than this. An append function can be called several times during a decode execution.

The buffer is in write mode when the append function is called and will automatically be set to read mode after the append function is finished. Do not use the `BufInitWriteMode` or `BufInitReadMode` macros in the append function.

If you want to add an append function for a buffer, then do the following steps:

1. Implement a C-procedure for appending data to the buffer. The C-procedure must have parameters compatible with `tBufAppend-Func`.

   ```
   typedef void (*tBufAppendBufFunc)(tBuffer buf,
   unsigned int len);
   ```

2. Use the macro `BufSetAppendBufFunc`, with the buffer and the append procedure as input parameters, immediately after a call to `BufInitBuf`.

Example 480: Buffer Append Procedure ----------

```
void MyAppendBuf( tBuffer buf,
                  unsigned int minbytes );
{
    /* receive at least minbytes from
       protocol or sockets or ... */

    BufPutSeg(buf,seg,len );
}
```

Setting appending function:

```
BufInitBuf(buf);
```

```
BufSetAppendBufFunc(buf,MyAppendBuf);
```

- `BufGetAppendBufFunc( buffer )`

  Returns a reference to an append function. Returns `NULL` if the append function does not set.

Small Buffer Implementation

A small buffer has a memory segment, an array of `unsigned char`. The encoded bit patterns are written to this segment and the bit patterns to decode are read from it.

The buffer pointer is a pointer to a control structure and a data structure. The control structure and the data structure contains information and memory pointers that the small buffer uses.

In the data structure, there is a pointer that points at the beginning of the memory segment, a pointer that points at the end of the memory segment and a pointer that points at current position for a read or write operation.

Memory is allocated inside the small buffer implementation. An initial memory segment is allocated. The size of the initial segment is set by defining the macro `CODER_SMALLBUF_SIZE`, which has the default value equal to 0x1000, see "Buffers configuration" on page 2896. When a memory segment is full and a bigger segment is needed, then a 2 times larger segment is allocated, the contents in the old segment copied and the old segment freed. The macros `CUCF_ALLOC` and `CUCF_FREE` are used for all memory allocation and de-allocation.

SDL Interface to Buffer Management System

SDL CoderBuf interface

In the Extended SDL Interface, encoded data is stored in the C-buffer interface. From SDL, this data is accessed via the SDL type CoderBuf. The interface is not a direct mapping of the C-interface. Some of the functions are not accessible from SDL. For more detailed information about C-interface to the buffer management system see "C interface to buffer management system" on page 2851.

Below is a summary of the contents in the file `CoderBuf.sdl`.

Types

The SDL buffer interface type names are the same as the ones in the C buffer interface (see "Types and definitions" on page 2851). The following type is introduced in the SDL buffer interface

- Type `ptr_tBMSUserMemory` is a pointer to `tBMSUserMemory`.

Operators

- `BufInitBuf : CoderBuf, tBMSBufType [-> int];`
- `BufInitBufWithMemory : CoderBuf, tBMSBufType, ptr_tBMSUserMemory [-> int];`
- `BufCloseBuf : CoderBuf [-> int];`
- `BufCloseBufToMemory : CoderBuf, ptr_tBMSUserMemory [-> int];`
- `BufInitReadMode : CoderBuf [-> int];`
- `BufInitWriteMode : CoderBuf [-> int];`
- `BufCloseReadMode : CoderBuf [-> int];`
- `BufCloseDeleteReadMode : CoderBuf [-> int];`
- `BufCloseWriteMode : CoderBuf [-> int];`
- `BufGetDataLen : CoderBuf -> tBMSLength;`
- `BufGetReadDataLen : CoderBuf -> tBMSLength;`
- `BufGetByte : CoderBuf -> unsigned_char;`
- `BufPeekByte : CoderBuf -> unsigned_char;`
- `BufPutByte : CoderBuf, unsigned_char;`
- `BufSetRule : CoderBuf, tERRule;`
- `BufGetRule : CoderBuf -> tERRule;`
- `BufCopyBuf : CoderBuf, CoderBuf [-> int];`
- `BufPutSeg : CoderBuf, Octet_string, int;`
- `BufGetSeg : CoderBuf, int -> Octet_string;`
- `BufPeekSeg : CoderBuf, int -> Octet_string;`

These operators are mapped to the C buffer interface functions, so the semantics of SDL operators is the same as described for the corresponding C procedures in "Buffer Management Functions" on page 2853.

SDL CustomCoderBuf interface

If you want a solution of your own for accessing the coder library from within SDL, you can use `CustomCoderBuf`. This can be achieved by modifying the file `CustomCoderBuf.sdl` and writing two C-procedures. `CustomCoderBuf` can then be used in the calls of `encode` and `decode` instead of `CoderBuf`.

If only minor changes are of interest, you could copy the `CoderBuf.sdl` file to `CustomCoderBuf.sdl` as a start. Even when the goal is to make a completely new access interface, `CoderBuf.sdl` could be used as an example.

Implementing the procedures `ASN1EncodeCustomBuf` and `ASN1DecodeCustomBuf` in a file of your own, means that the file should then be included in the build process. For further information about parameters and return values of those procedures see the file `cucf_er_sdt.h` in the coder library. When `CustomCoderBuf` is selected in the user interface, the code generator generates code that calls those two procedures.

The procedures `ASN1EncodeCoderBuf`/`ASN1DecodeCoderBuf` and `ASN1EncodeOctet`/`ASN1DecodeOctet` could be seen as SDL Suite versions of a CustomCoderBuf. The first parameter in that implementation is `CoderBuf` and `Octet_string` respectively. When implementing your own `CustomCoderBuf`, you can select whatever type of this first parameter you want. However, it is important that the type defined in `CustomCoderBuf.sdl` file corresponds to the first parameter of the two above procedures.

The choice of using a custom implementation must be selected in the Analyzer dialog. See "Analyze SDL" on page 113 in chapter 2, *The Organizer*.

User defined Buffer Handling

The coder library can be configured to work with user defined buffers. It is possible to write your own specific buffer handling procedures which will be invoked instead of the default buffer handlers.

All coder library buffer implementations are based on the open buffer interface approach which allows you to introduce user types of buffer management in the same style as, for example, predefined small buffer management implementation.

If you want to use your own buffers, then perform the following steps:

1. Define the structure `UserBufFuncs` of type `tBaseBuf` (see "Types and definitions" on page 2851).
2. Set the value `bms_UserBuffer` to the field `BufType`.
3. Implement the buffer access functions and assign references to these functions to the corresponding fields of `UserBufferFuncs`. Buffer access functions should have compatible interfaces with the function pointer types defined for `tBaseBuf` structure in the file `bms.h` (see "Buffer Management System sub-directory" on page 2889) and should support the semantics of buffer access functions described in "Buffer Management Functions" on page 2853.

Note: Buffer implementation example

The small buffer implementation is based on the open buffer interface and can be used as an example of a user defined buffer implementation. Looking into the files `bms_small.h` and `bms_small.c` in the installation (see "Buffer Management System sub-directory" on page 2889) will be helpful before starting the implementation.

4. Compile coder library with `CODER_USE_USERBUF` compile switch, see "Buffers configuration" on page 2896,
5. Compile and link the application with the user buffer handling functions and the defined structures.

Memory Management System

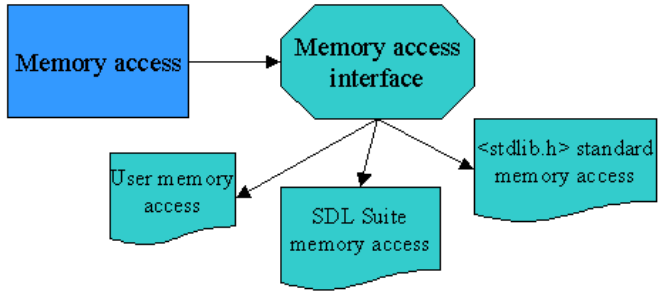


Figure 511: Memory management

The coder solution contains memory management functions. There are only two basic functions used for memory management: alloc and free.

Memory management in the coder library consists of allocation and freeing of memory, and releasing of memory in case of errors.

Coder library contains two levels of memory access functions:

- Pure allocation and freeing of memory - `CUCF_ALLOC(size, type)` and `CUCF_FREE(ptr, size, type)`,
  - where `size` is of type `size_t` and represents the size of memory in bytes to be allocated,
  - `type` is not used (reserved for the future),
  - `ptr` is of type `void*` and is a pointer to the memory to be released.
- Safe memory allocating and releasing - `CUCFAlloc(info, size, type)`, `CUCFFree(info, ptr, size, type)` and `CUCFRelease(info, ptr)`, where `size`, `type` and `ptr` are the same as for the previous memory handling functions. Apart from allocating, `CUCFAlloc` function saves a pointer to the allocated memory block in the internal stack, which is used for memory freeing in case of error. The first parameter `info` in these three memory handling functions is of type `tEMSInfo*` and it is a pointer to the stack where all

allocated memory segments are stored. Safe functions use pure memory management functions for memory operations. `CUCFAlloc` calls `CUCF_ALLOC` for allocating memory for storage cell in the stack and for allocating `size` required bytes, `CUCFFree` frees memory pointed by `ptr` and also the corresponding stack cell.

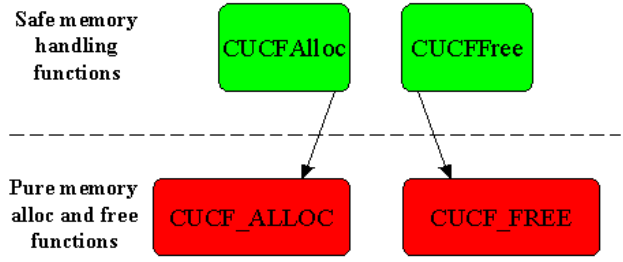


Figure 512: Memory management functions

Safe memory handling functions are used in those places where memory freeing procedure will not be performed in case of error, for example, in the coder library kernel. Pure memory handling functions are called from the safe functions. They are also called directly in those places where memory can still be released even after an error has occurred, for example, buffer memory allocation (memory is released by BufCloseBuf function).

Predefined Memory Handling

There are two types of predefined memory handling functions which can be used to perform allocation and freeing of memory:

- SDL Suite memory handlers `XALLOC` and `XFREE`, see "Functions for Allocation and Deallocation" on page 3117 in chapter 61, *The Master Library*, and
- standard `malloc` and `free` functions from `<stdlib.h>` which are the default memory handlers.

Users can also define their own application specific memory handling functions, see "User defined Memory Handling" on page 2868.

The type of memory handler to be invoked for allocation and freeing of memory is configured when compiling coder library by compile switches, see "Memory management configuration" on page 2897.

User defined Memory Handling

The coder library can be configured to work with user-specific memory handling procedures. It is possible to write your own allocate and free procedures, which will be invoked instead of the default memory handlers.

> **Note: Performance optimization**
>
> Fast memory handling, if applicable, can help optimize the performance of the executable quite a lot. One of the examples of fast memory handling is working with statically allocated memory instead of dynamically allocated in alloc and free procedures.

If you want to use your own memory handler, perform the following steps:

1. Create file `mms_user.h` and insert definitions of macros `USER_ALLOC_FUNC(size, type)` and `USER_FREE_FUNC(ptr, size, type)` to point to the user defined memory handling procedures. These macros will be used for redefining pure memory handling procedures.
2. Compile the coder library with the `CODER_MMS_USER` compile switch, see "Memory management configuration" on page 2897.
3. Add include path for the file `mms_user.h` to the compilation settings.
4. Compile and link the user memory handler function to the application.

Example 481: File mms_user.h ——————————————————

```
#ifndef mms_user_h
#define mms_user_h

#define USER_ALLOC_FUNC(size,type)  UserAlloc(size)
#define USER_FREE_FUNC(ptr,size,type) UserFree(ptr)

extern void* UserAlloc(size_t size);
extern void UserFree(void* ptr);
```

```
#endif
```

There are also two BMS user-specific memory handling procedures that are referenced through macros defined in the `mms_user.h`:

```
#define USER_BMS_ALLOC_FUNC(buffer, size, type)
UserBmsAlloc(buffer, size)
#define USER_BMS_FREE_FUNC(coder, ptr, size, type)
UserBmsFree(buffer, ptr, size)
```

These interfaces are used only in the buffer functions (see "Buffer Management Functions" on page 2853). By default (without this definition) `USER_ALLOC_FUNC` and `USER_FREE_FUNC` are used inside the buffer functions.

The BMS user-specific memory handlers use buffer access types `tCoder*` or `tBuffer` as the first argument. This argument opens the access to the `tUserData` (see "User Data" on page 2884) inside the BMS memory handling:

```
void* UserBmsAlloc(tCoder* Coder, size_t Size)
{
  tUserData* data = BufGetUserData(Coder);
  ... /* user implementation */
}

void UserBmsFree(tCoder* Coder, void* Ptr, size_t
Size)
{
  tUserData* data = BufGetUserData(Coder);
  ... /* user implementation */
}
```

Error Management System

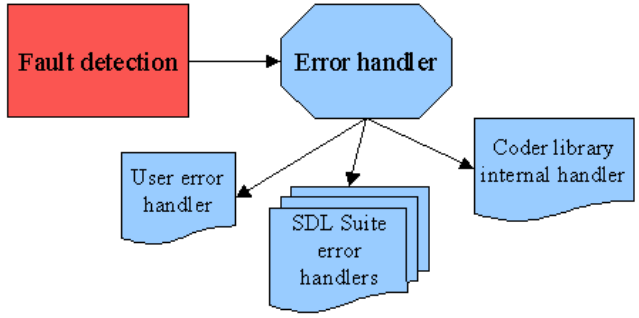


Figure 513: Error management

The coder solution contains error management functions. The error management consists of fault detection, error handling and error output. The error management is optional and can be removed from the application.

The fault detection consists of monitoring code that checks for all kinds of faults when encoding and decoding. When a problem is detected the fault detection part will start the error handler. The monitoring code is controlled by C macros and can be removed by defining macros, see "Error checking configuration" on page 2893. The user defined error handling functionality is described in "User defined Error Handling" on page 2880.

SDL Suite error output functionality has several error output implementations for the different types of kernels. The behavior is different for simulation and for application. It is possible to register a different error output function. See "User defined Error Output" on page 2883.

There are several types of error output functions which can be used to perform error output:

- SDL Suite error output as default
- Coder library error output
- user defined application specific error output functions.

The type of error output to be invoked upon error detection is configured when compiling coder library by compile switches, see "Error output configuration" on page 2897.

Error Management Interface

Encode and decode functions return a status code, which is equal to a success code, or one of the error codes in case of failure. Error codes are visible in both SDL and C functional interfaces. If an error occurs, its code is returned by the encoding or decoding function, and is written into the buffer at the same time.

The behavior of the fault detection can be controlled by compilation switches, that are described in "Error checking configuration" on page 2893. For instance, it is possible to remove all fault detection by defining a corresponding switch.

Detailed error messages

Error messages contain a type name with a path to the field where an error has occurred. The path contains a sequence of SET/CHOICE/SEQUENCE fields and open type objects from the root type up to the field where an error has occurred.

There are two possible representations for the path:

- Path fields are represented by their names. This representation is available only if one of the following compile switches are present, `CODER_TI_NAMES` (see "Error output configuration" on page 2897) or `CODER_VMS_PRINT` (see "Printing configuration" on page 2899). For example:

  ```
  ERROR 21 No more data for read at BOOLEAN, type:
  BOOLEAN, path: yASN1_Type1.field2.comp1
  ```

- Path fields are represented by their indexes - in this case root type is named "undef" and path contains indexes of SEQUENCE/SET components, CHOICE alternatives and open type objects, for example:

  ```
  ERROR 21 No more data for read at BOOLEAN, type:
  undef, path: SEQUENCE.2.1
  ```

By default the second representation with indexes is used.

If an error occurs within the encoding/decoding then you can unambiguously understand the exact place of the error. For example if we have a complex type with the several nested sequences:

```
Type1 ::= SEQUENCE {
  field1 Type2,
  field2 Type2
}
Type2 ::= SEQUENCE {
  comp1 [0] Bool,
  comp2 [1] Int
}
Bool ::= [0] BOOLEAN
Int ::= [1] INTEGER
```

And the error is:

```
ERROR 21 No more data for read at BOOLEAN, type:
BOOLEAN, path: yASN1_Type1.field2.comp1
```

(in the first representation)

```
ERROR 21 No more data for read at BOOLEAN, type:
undef, path: SEQUENCE.2.1
```

(in the second representation)

Then the exact place of the error is known. The error occurred in `Bool` type in the second nested sequence by the path `Type1.field2.comp2`.

The benefits of this feature become significant in a huge ASN.1 specifications with the long nested fields, like UMTS. This is an example of a real error message from the RRC specification (subpart of UMTS):

```
ERROR 22: Wrong TAGGED identifier octet, type:
CPCH_SetInfo_INLINE_3, path: yASN1_DL_DCCH_Message.mes-
sage.radioBearerSetup.modeSpecificPhysChIn-
fo.fdd.cpch_SetInfo.ap_AICH_ScramblingCode
```

Detailed error messages (type and path) is used in the `ec_VAL_` and `ec_DEC_` errors.

The default configuration includes detailed error messages. Detailed messages require extra resources from the speed and from the Type Info size (if names information is enabled). The encoding/decoding speed becomes a bit slower because the feature requires extra manipulations with the stack to store the temporary path. If you do not need to use this feature then you should use `CODER_REMOVE_PATH` compile switch (see "Error handling configuration" on page 2898). Error messages in this case will look like:

```
ERROR 21 No more data for read at BOOLEAN, type:
undef, path: undef
```

SDL error interface

In SDL, error codes are represented by integer values and are defined as Integer synonyms. Names of synonyms are the same as error codes described in "Error codes" on page 2873. The file `codererrors.sdl` contains a complete list of Integer synonyms for coder error codes. This file must be included to the SDL system that is going to check coder error codes after encoding and decoding.

C error interface

In the C error interface, error codes are literals of one big enumerated type, and they can be treated as C integer values. All possible error codes are listed in the file `errors.h` in the 'ems' folder of the library installation (see "Files and File Descriptions" on page 2886). For detailed description of error codes see "Error codes" on page 2873.

Error codes

If the decode or encode succeeded, `ec_SUCCESS` is returned.

In case of failure, an error code is returned that helps to understand the reason of the failure. All possible error codes with short descriptions are listed below.

Memory access errors

ec_MEM_NotEnoughMemory

The application has run out of memory and one of the coder library functions can not allocate memory.

Buffer errors

ec_BUF_DifferentBufferTypes

This is a buffer copying error. The `BufCopyBuf` function belongs to the general buffer interface which supports different buffer types. The error occurs when the source and destination buffers in the copying function are of different buffer types.

ec_BUF_WrongBufferType

The message is reported when the buffer initialize function has been called with the buffer type which has been excluded from the application by compile switches (see "Buffers configuration" on page 2896).

ec_BUF_CloseNotInitialized

The message is reported when `BufCloseBuf` is called on a buffer that has not been initialized.

ec_BUF_WorkWithNotInitialized

One of the two buffer modes, read or write, has to be initialized before accessing the buffer with reading or writing procedures. If a read or write procedure has been applied to a buffer that has not been initialized, the ec_BUF_WorkWithNoneInitialized error is reported.

ec_BUF_NullPtrToUserMemory

BufInitBufWithMemory or BufCloseBufToMemory is called with `NULL` pointer to user memory.

ec_BUF_NotEnoughUserMemory

This error is specific for buffers that are initialized by the user memory, and it denotes that the size of the user memory assigned to the buffer is not enough for storing data.

ec_BUF_OpenOpened

This error is reported when trying to reopen the buffer which has already has been opened for write or read mode, and has not been closed afterwards.

ec_BUF_IllegalClose

This error is reported when BufCloseBuf is called on the buffer for which one of the modes, reading or writing, has not been closed.

ec_BUF_CloseWrongMode

This error is returned when trying to close the buffer for a mode different from the opened one. This situation can happen, for example, when BufCloseWriteMode follows the function BufInitReadMode, or when

BufCloseReadMode is called to the buffer which mode has been initialized by the function BufInitWriteMode.

ec_BUF_OperationWrongMode

This error denotes that you are trying to perform an incorrect operation for the current buffer mode. For example, you are trying to read data from the buffer which has been opened for writing by function BufInitWriteMode.

ec_BUF_NoMoreDataForRead

This error is returned when you are trying to read data from a buffer which is already empty.

ec_BUF_TooBigNumberOfBits

The error is returned when you are trying to put or get more bits from the buffer than it is possible to pass to the bit buffer interface function. The operations BufGetBits and BufPutBits operate with bits stored in the value of type long and can not process more then sizeof(long) number of bits. So when they are called with a number of bits more then sizeof(long), an error message is reported.

ec_BUF_InvalidEncodingRules

This error code is returned when the combination of bit masks passed as the second parameter to the function BufSetRule does not form a valid encoding rule. For example, `BufSetRule( buffer, er_BER | er_Aligned )` will return this error code because `er_Aligned` is an alternative of PER and does not have any sense together with BER.

Value errors

ec_VAL_IllegalRealBase

The real value received for encoding is represented with a base that cannot be handled by the encoder function.

ec_VAL_WrongConstrainedValue

The value passed to the coder function does not satisfy constraints specified for the corresponding type.

ec_VAL_WrongConstrainedLength

The size of the value passed to the coder function does not satisfy the size constraint specified for the corresponding type.

ec_VAL_WrongConstrainedAlphabet

Characters in the string value passed to the coder function do not satisfy the permitted alphabet constraint specified for the corresponding type.

Decoding errors

ec_DEC_NoMoreDataForRead

This error is returned when you are trying to read data from a buffer which is already empty. There is a difference in the formats between `ec_DEC_NoMoreDataForRead` and `ec_DEC_NoMoreDataForRead`. `ec_DEC_NoMoreDataForRead` uses the detailed error message approach (with type and path) but `ec_BUF_NoMoreDataForRead` does not.

ec_DEC_WrongIdentifierOctet

This is a BER specific error, based on BER encoding TLV (Type-Length-Value) structure. The type information from TLV is stored in a so called identifier octet. If the data in the identifier octet for the sequence of bits to be decoded is wrong (for example, ASN.1 type is defined to be INTEGER, but the identifier octet for the value of that type claims that value to be decoded if of type BOOLEAN), an error is returned.

ec_DEC_WrongLength

For some ASN.1 types, the range of the encoded value length is restricted. When length is out of range, the wrong length error is returned.

ec_DEC_TooBigTagNumber

Tag numbers are stored in the variable of type unsigned long, so the value of a tag is restricted. When the tag number occupies more then 32 bits, restriction is violated and this error message is returned.

ec_DEC_WrongConstructedLengthPrefix

When constructed encoding is used, each group of bytes from the encoding is prefixed by the group information. For example, if it is the last group in the constructed encoding or not, and similar. Constructed encoding prefix occupies 8 bits, but not all bit combinations are meaningful. When the combination is wrong, this error is returned.

ec_DEC_WrongUnusedBits

This is a BER specific error. BIT STRING values in BER are encoded into a sequence of bytes, although they cannot contain exactly an integer number of 8 bits. For correct value encoding, the number of unused bits in the last byte is also encoded. When this number is not in the range between 0 and 7, an error is returned.

ec_DEC_TooBigSubIdent

This object identifier decoding error is reported when the sub-identifier is too big and cannot be stored in the variable of type long (4 bytes).

ec_DEC_WrongRealPrefix

The real value encoding contains a prefix denoting the type of real value that has been encoded, plus infinity and minus infinity real values, zero real value, other common real value (mantissa + base + exponent). The real prefix is encoded into 8 bits but not all of them have meaning. If the real encoding contains a bad prefix, this error is reported.

ec_DEC_UnsupportedRealBase

Trying to decode a real value which has been encoded with a base that is not supported in the coding library.

ec_DEC_UnsupportedRealDecimalEncoding

Decimal encoding of real values is not supported. If real value has been encoded according to decimal real encoding algorithm, the decoder will not be able to decode it and it will return this error code.

ec_DEC_RequiredComponentIsAbsent

This error can be reported when decoding SEQUENCE or SET type values and it denotes that not all required components are present in the value encoding.

ec_DEC_ExtRequiredComponentIsAbsent

This is SEQUENCE and SET decoding specific error. It points out that not all required components from extension addition group are present in the value encoding, although the group itself is encoded as present.

ec_DEC_AbsentComponentIsPresent

This error can appear while decoding a value of sequence or set type with the ABSENT constraint applied. When the component constrained to be absent is present in the encoded value, this error message is reported.

ec_DEC_AbsentAlternativeIsPresent

This error can appear while decoding a value of choice type with the ABSENT constraint applied. When the alternative constrained to be absent is present in the encoded value, this error message is reported.

ec_DEC_UnknownComponent

This is SEQUENCE and SET specific decoding error, and it is reported when the encoded field tag is not a known tag from the type fields of SEQUENCE or SET type.

ec_DEC_UnknownAlternative

This is CHOICE specific error, and it is returned when the encoded choice alternative has got a tag which does not belong to the set of possible tags for the decoded choice type.

ec_DEC_NoOpenId

This is open type specific error. It is reported when open type value can not be decoded because open type identifier field is absent for some reason in the encoding.

ec_DEC_UnknownObject

This is open type specific error. It is reported when decoded open type identifier is not equal to any of identifier in the open type restricting table.

ASCII decoding errors

Each ASCII decoding error refer to a particular type and denotes that there are problems or errors when decoding value of that type, for example, ec_ERROR_DECODING_PARSTART is reported when there are errors when decoding start of signal parameter or variable from the buffer.

All possible ASCII decoding errors are listed below.

ec_ERROR_DECODING_INTEGER
ec_ERROR_DECODING_REAL
ec_ERROR_DECODING_BOOLEAN
ec_ERROR_DECODING_TIME
ec_ERROR_DECODING_CHARSTRING
ec_ERROR_DECODING_BIT
ec_ERROR_DECODING_BITSTRING
ec_ERROR_DECODING_OCTET
ec_ERROR_DECODING_OCTETSTRING
ec_ERROR_DECODING_SIGNALID
ec_ERROR_DECODING_STRUCT
ec_ERROR_DECODING_CHOICE
ec_ERROR_DECODING_POWERSET
ec_ERROR_DECODING_BAG
ec_ERROR_DECODING_STRING
ec_ERROR_DECODING_ARRAY
ec_ERROR_DECODING_REF
ec_ERROR_DECODING_USERDEF
ec_ERROR_UNKNOWN_TYPE_NODE
ec_ERROR_DECODING_PARSTART
ec_ERROR_DECODING_PAREND

Internal errors

ec_INT_InternalError

This is coder library internal error. If coder function returns this error, please, contact IBM Rational Customer Support. Internal error will never be reported if `CODER_CHECK_NONE_INNER` compile switch is set up (see "Error checking configuration" on page 2893), in this case all internal error checks are removed from the library code.

ec_INT_UnsupportedType

This is also coder library internal error. It is reported when the type specified for the encode or decode function in the ASN.1 type info structure is not supported by the coder library. Internal error will never be reported if `CODER_CHECK_NONE_INNER` compile switch is set up (see "Error checking configuration" on page 2893), in this case all internal error checks are removed from the library code.

> **Note: Error codes backwards compatibility**
>
> Backwards compatibility is implemented for those error codes which have been removed or renamed in comparison with the previous versions of the coder library error management system. All old error names will be mapped to new ones with the same functionality for the systems using previous version of coder library. An example of error code which is not used in the new coder library any more is `ec_SMLBUF_READ_ERROR`.

Definition of your own error codes

It is possible to specify your own error codes. It might be useful for user encoding rules or user memory management implementation.

If you want to use your own error codes, perform the following steps:

1. Create file `errors_user.h` and insert your own error codes by using the following format:

CUCF_ERROR(ec_<error identifier>, "Error message")

Example 482: File errors_user.h ----------

```
CUCF_ERROR(ec_MY_ERROR_ONE, "My error one")
CUCF_ERROR(ec_ME_ERROR_TWO, "My error two")
```

----

2. Compile the coder library with the `CODER_EC_USER`="errors_user.h".
3. Add include path for the file `errors_user.h` to the compilation settings.

User defined Error Handling

This functionality allows to call a user defined error handling function inside the decoding procedure. If an error occurs this function will be

called. Combined with the `BufGetErrorPath( buffer )` buffer interface (see "Buffer Management Functions" on page 2853), error handling function gives an opportunity to insert a default value to the field for which the decoding failed.

An error handler function must be written by the user according to the following format:

```
typedef void (*tBufErrInitFunc)(tBuffer Buffer);
```

There is an interface in the buffer functionality `BufSetErrInitFunc( buffer, func )` that allows to set user error handling by reference into the buffer and it will be used by the decoder. `BufGetErrInitFunc( buffer )` returns a reference to an error handling function. By default there is no error handler and it returns `NULL`. Below there is an example with the user defined error handling.

These two checks between ErrorPath and TypeInfo help to determine the exact place where an error has occurred:

```
#define EQ_TI(field, ti) \
  ((tASN1TypeInfo*)(field)==(tASN1TypeInfo*)&ti)
#define EQ_CMP(field, ti) \
  (((tASN1Component*)(field))->TypeInfo==(tASN1TypeInfo*)&ti)
```

Error handling:

```
void TinyErrorHandling(tBuffer b)
{
  int res = BufGetErrorCode(b);
  tErrorPath *ep = BufGetErrorPath(b);
  LineB* lineB = (LineB *)BufGetMainVal(b);
  if ( res == ec_VAL_WrongConstrainedValue &&
       ep->NumOfFields == 2 &&
       EQ_TI(ep->Fields[0], yASN1_LineB) &&
       EQ_CMP(ep->Fields[1], yASN1_PointB_INLINE_0))
    {
      lineB.point1.x = 40;
      return;
    }
  if ( res == ec_VAL_WrongConstrainedValue &&
       ep->NumOfFields == 2 &&
       EQ_TI(ep->Fields[0], yASN1_LineB) &&
       EQ_CMP(ep->Fields[1], yASN1_PointB_INLINE_1))
    {
      lineB.point2.y = 25;
      return;
    }
}
```

ASN.1:

```
Tiny DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
  PointA ::= SEQUENCE {
    x         INTEGER (0..42),
    y         INTEGER (0..26)
  }
  PointB ::= SEQUENCE {
    x         INTEGER (0..40), -- lineA.point2.x = 42
    y        INTEGER (0..25)  -- lineB.point1.y = 26
  }
  LineA ::= SEQUENCE {
    point1   PointA,
    point2   PointA
  }
  LineB ::= SEQUENCE {
    point1   PointB,
    point2   PointB
  }
  lineA LineA ::= { point1 { x 11, y 26 }, point2 {
x 42, y 24 } }
  lineB LineB ::= { point1 { x 11, y 25 }, point2 {
x 40, y 24 } }
END
```

The code example with the encoding of `LineA` type and `LineB` as the type for the decoding. The encoded `lineA` value does not satisfy the `LineB` constraint and that is why there are two constraint errors:

```
ERROR 18 Wrong INTEGER constrained value, type:
PointB_INLINE_0, path: yASN1_LineB.point1.x
ERROR 18 Wrong INTEGER constrained value, type:
PointB_INLINE_1, path: yASN1_LineB.point2.y
```

The `TinyErrorHandling` function catches these errors with a corresponding assignment to the correct values.

```
BufInitBuf(b, bms_SmallBuffer);
BufSetErrInitFunc(b, TinyErrorHandling); /* set
TinyErrorHandling as decoding
BufInitWriteMode(b);
res = ASN1_ENCODE(b, (tASN1TypeInfo *)&yASN1_LineA,
lineA);
BufCloseWriteMode(b);
BufInitReadMode(b);
res = ASN1_DECODE(b, (tASN1TypeInfo *)&yASN1_LineB,
dec_lineB); /*
BufCloseReadMode(b);
BufCloseBuf(b);
```

User error handling requires the same resources as for detailed error messages (see "Detailed error messages" on page 2871). The encoding/decoding speed becomes slower because of some extra manipula-

tions with the stack. CODER_REMOVE_PATH compile switch (see "Error handling configuration" on page 2898) should be defined to disable the error handling.

User defined Error Output

It is possible to write your own error output function, which will be invoked when a fault is detected.

If you want to use your own error output, perform the following steps:

1. Implement a C-procedure for error output with the prototype `void USERErrorOutputFunc(FILE* File, tEMSErrorCode Code, va_list MessageArguments)`. `tEMSErrorCode` is an enumerated type containing all possible error codes returned by coder library functions, see "Error codes" on page 2873. You can also find a list of the different error codes in `errors.h` from `/cucf/ems` folder.
2. Compile the coder library with CODER_EO_USER compile switch, see "Error output configuration" on page 2897,
3. Compile and link the user error output function to the application.

The `USERErrorOutputFunc` function takes error message arguments as the third parameter which corresponds to the message string defined in the coder library. The error management system provides a function that allows you to get the coder library message string for the error code: `char* CUCFGetErrorMessage(tEMSErrorCode Code)`.
Example 483 illustrates one possible way of defining a user error output function.

Example 483 Error output

```
void USERErrorOutputFunc( FILE* File,
                          int Code,
                        va_list MessageArguments )
{
  /* user error output with the code and message */
  fprintf(File, "USER ERROR %d ", Code);
  vfprintf(File, ErrorMessageArray[Code],
MessageArguments);
  fputc('\n', File);
}
```

User Data

There is an option to include your own data fields into the buffer type. C-type `tUserData` is defined in the `user_data.h` file and it can be included into the buffer by `CODER_USER_DATA` compile switch (see "User data configuration" on page 2898). By default type `tUserData` is defined as `void*`, but you can redefine it according to your needs.

`tUserData` with your own fields might be useful in the user error handling function (see "User defined Error Handling" on page 2880) to store the data for the later manipulation outside the decoder or in the `USER_BMS_ALLOC_FUNC`/`USER_BMS_FREE_FUNC` (see "User defined Memory Handling" on page 2868). The buffer interface `BufGetUserData( buffer )` (see "Buffer Management Functions" on page 2853) returns a reference to `tUserData` for your further handling. If you want to use your own data fields as `tUserData`, perform the following steps:

1. Create file `user_data.h` and insert declaration of the `tUserData` C-structure.
2. Compile the coder library with the `CODER_USER_DATA` compile switch.
3. Add include path for the file `user_data.h` to the compilation settings.

Example 484: File user_data.h ----------

```
#ifndef user_data_h
#define user_data_h

typedef struct {
  int UserInt1;
  int UserInt2;
  int UserInt3;
} tUserData;

#endif
```

Printing Opportunities

The print functions can be used for test and debug purposes. They will be available at runtime only if the compilation switch CODER_VMS_PRINT is set, see "Printing configuration" on page 2899.

There are two print functions available, ASN1_PRINT_TYPE and ASN1_PRINT. The print functions use the type information in the same way as an encoding function.

The function ASN1_PRINT_TYPE prints the contents of the type information structure for one particular type. There are two input parameters, the first is a file handle and the second is a reference to the type information.

The function ASN1_PRINT prints the value of a variable or signal parameter. This print function has got three input parameters, the first is a file reference, the second is a reference to the type information structure and the third is a reference to the variable or the signal parameter.

Example 485: Print type information

```
FILE * logfile;

logfile = fopen( "asn1print.log", "w" );

ASN1_PRINT_TYPE( logfile,
                 (tASN1TypeInfo *)&yASN1_TestType);

fclose( logfile );
```

Example 486: Print value of signal parameter

```
FILE * logfile;

logfile = fopen( "asn1print.log", "w" );

ASN1_PRINT( logfile, (tASN1TypeInfo *)&yASN1_TestType,
     (void *)&((yPDef_outsig *)(*SignalOut))->Param1);
fclose( logfile );
```

Structure and Configuration

An application with ASN.1 encoding or decoding support contains the following:

- Encoding and decoding functions. One set of functions for BER encoding and decoding and one set of functions for PER encoding and decoding.

> **Note: ASCII coder**
>
> ASCII encoding and decoding, which is based on SDL Suite internal encoding rules is also available. ASCII encoding and decoding is described in "SDL Data Encoding and Decoding, ASCII coder" on page 2790 in chapter 57, *Building an Application*.

- Buffer management functions.
- Error management functions (optional).
- Print functions (optional).
- ASN.1 type information generated by the ASN.1 Utilities.
- SDL type and operator information generated by the SDL C code generators.

Files and File Descriptions

This section describes the static file structure of the coder directories in the installation.

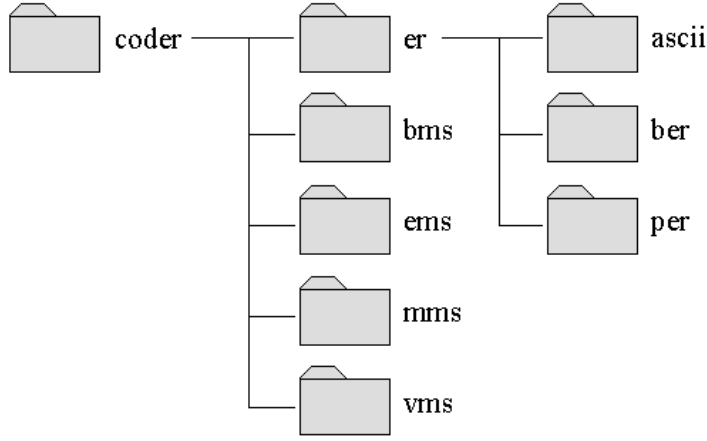


Figure 514: The coder directory

The coder directory contains five sub-directories which correspond to the functional modules in the coder library:

- `coder/er` - this directory contains three sub-directories which correspond to different encoding rules supported by the coder library.
- `coder/bms` - Buffer Management System files.
- `coder/ems` - Error Management System files.
- `coder/mms` - Memory Management System files.
- `coder/vms` - Value Management System files.

There are also several files included to the root coder directory:

- `cucf.h` - main header file of the coder library. It includes all useful header files from the library functional modules.
- `cucf_cfg.h` - standard C library includes, like `<stdio.h>`, `<sdtlib.h>` and others.
- `cucf_er.h` - declarations of main encode and decode coder interface functions.
- `cucf_er.c` - definition of main encode and decode coder interface functions.

- `cucf_er_sdt.h` - declarations of SDL Suite specific encode and decode interface functions which are used for encode and decode within SDL support.
- `cucf_er_sdt.c` - definition of SDL Suite specific encode and decode interface functions which are used for encode and decode within SDL support.
- `cucf_er_ttcn.h` - declarations of TTCN Suite specific encode and decode interface functions which are used for encode and decode within TTCN support.
- `cucf_er_ttcn.c` - definition of TTCN Suite specific encode and decode interface functions which are used for encode and decode within TTCN support.
- `coderascii.h`, `coderber.h`, `coderper.h`, `coderucf.h` - these files are not used inside the library, but provide backwards compatibility solutions for the SDL systems created with the previous versions of SDL coder support.

Encoding rules sub-directory

Sub-directory `coder/er` contains three sub-directories corresponding to different types of encoding rules supported by the coder library:

- `ascii`

  This directory contains files with encoding and decoding function according to internal rules of the SDL Suite. The ASCII coders are described in "SDL Data Encoding and Decoding, ASCII coder" on page 2790 in chapter 57, *Building an Application*.

- `ber`

  This directory contains files with encoding and decoding functions according to BER (Basic Encoding Rules), see "BER sub-directory" on page 2888.

- `per`

  This directory contains files with encoding and decoding functions according to PER (Basic Encoding Rules), see "PER sub-directory" on page 2889.

BER sub-directory

Sub-directory `coder/er/ber` contains the following files:

- `ber.h` - header file with declaration of `BEREncode` and `BERDecode` functions.

- ber_base.h - contains declarations of basic functions needed by the encode and decode functions.
- ber_base.c - contains implementation of basic functions needed by encode and decode functions.
- ber_content.h - contains declarations of functions that encode and decode ASN.1 predefined types.
- ber_content.c - contains implementation of functions that encode and decode ASN.1 predefined types.
- ber_decode.c - definition of main outermost function BERDecode which references functions from ber_base and ber_content modules.
- ber_encode.c - definition of main outermost function BEREncode which references functions from ber_base and ber_content modules.

PER sub-directory

Sub-directory coder/er/per contains the following files:

- per.h - header file with declaration of PEREncode and PERDecode functions.
- per_base.h - contains declarations of basic functions needed by the encode and decode functions.
- per_base.c - contains implementation of basic functions needed by encode and decode functions.
- per_content.h - contains declarations of functions that encode and decode ASN.1 predefined types.
- per_content.c - contains implementation of functions that encode and decode ASN.1 predefined types.
- per_decode.c - definition of main outermost function PERDecode which references functions from per_base and per_content modules.
- per_encode.c - definition of main outermost function PEREncode which references functions from per_base and per_content modules.

Buffer Management System sub-directory

Sub-directory coder/bms contains the following files:

- bms.h - base buffer management declarations, this header contains base buffer control structure definitions. This is the main header file for buffer management.

- `bms.c` - implementation of base buffer functions which are common for all types of buffers.
- `bms_small.h` - small buffer management specific declarations.
- `bms_small.c` - implementation of small buffer management.

Error Management System sub-directory

Sub-directory `coder/ems` contains the following files:

- `ems.h` - error management declarations. This is the main header file for coder error handling.
- `ems.c` - implementation of common error management functions.
- `ems_eo_sdt.h` - SDT specific error management declarations and includes.
- `ems_eo_sdt.c` - implementation of SDT specific error handling functions.
- `errors.h` - declaration of error codes and error information.

Memory Management System sub-directory

Sub-directory `coder/mms` contains the following files:

- `mms.h` - declarations of pure memory handling macros `CUCF_ALLOC` and `CUCF_FREE` and safe memory management functions `CUCFAlloc` and `CUCFFree` (see "Memory Management System" on page 2866). This is the main header file for coder memory handling.
- `mms.c` - implementation of safe memory handling functions.

Value Management System sub-directory

Sub-directory `coder/vms` contains the following files:

- `vms.h` - this is the main header file for value management system, it contains includes of all other useful header files from this folder.
- `vms_type.h` - definitions of types used in ASN.1 type nodes.
- `vms_macro.h` - definitions of macros which are used for declaration and filling ASN.1 type structure nodes. Calls to these macros are generated by ASN.1 coder generator.
- `vms_vr_sdt.h` - definitions of macros which are used for inserting SDL Suite specific information to ASN.1 type structure nodes. Calls to these macros are generated by ASN.1 coder generator only for SDL Suite coder generation.
- `vms_vr_ttcn.h` - definitions of macros which are used for inserting TTCN Suite specific information to ASN.1 type structure nodes.

Calls to these macros are generated by ASN.1 coder generator only for TTCN Suite coder generation.

- `vms_export.h` - type node declarations for predefined ASN.1 types, such as `BOOLEAN`, `INTEGER`, `BIT STRING`, `OCTET_STRING`, `NULL`, `OBJECT IDENTIFIER`, `REAL`, string types.
- `vms_export.c` - type node definitions for predefined ASN.1 types.
- `vms_internal.h` - internal value representation type and macro definitions.
- `vms_base.h` - declarations of internal value representation access procedures.
- `vms_base.c` - implementation of internal value representation access procedures.
- `vms_check.h` - declarations of check procedures.
- `vms_check.c` - implementation of procedures that perform error checking of type nodes and values.
- `vms_print.h` - declarations of print procedures used for test and debug purposes.
- `vms_print.c` - implementation of print procedures used for test and debug purposes.

Compilation switches

You can change the default properties and configure the BER and PER coders by setting the compilation switches. Some of these switches can be set by choosing options in the Targeting Expert, see chapter 59, *The Targeting Expert*.

Available compile switches can be separated onto several groups. Below there are descriptions of compile switches that can be used to configure the behavior of coder functions.

Encoding rules configuration

Removing runtime encoding rules availability

- CODER_REMOVE_ASCII - compile coding library without ascii encoding rules available at runtime
- CODER_REMOVE_BER - compile coding library without BER encoding rules available at runtime

- CODER_REMOVE_PER - compile coding library without PER encoding rules available at runtime
- CODER_USE_UER - add user defined encoding and decoding functions to be available at runtime (file “uer.h“ will be used during library compilation)

By default all the library included ascii, ber and per encoding rules to be present in the compiled object library.

Choosing default encoding rules

This compilation switches influence on which encoding rules will be applied when the coder function is called without specifying which type of encoding rules should be applied

- CODER_ER_DEFAULT_PER - use PER as default encoding rules
- CODER_ER_DEFAULT_BER - use BER as default encoding rules
- CODER_ER_DEFAULT_UER - apply USER encoding rules by default

The default setting is CODER_ER_DEFAULT_BER.

- CODER_BER_DEFINITE - use definite length form encoding for BER by default
- CODER_BER_INDEFINITE - use indefinite length form encoding for BER by default

The default setting is CODER_BER_INDEFINITE.

- CODER_PER_NO_ENDPAD - use PER unaligned without end padding as default encoding rules
- CODER_PER_ALIGNED - use PER aligned as default encoding rules
- CODER_PER_UNALIGNED - use PER unaligned as default encoding rules

The default setting is CODER_PER_UNALIGNED.

- CODER_BER_CONSTRUCTED - use constructed form of BER by default

- CODER_BER_PRIMITIVE - use primitive form of BER by default

The default setting is CODER_BER_PRIMITIVE.

- CODER_BER_CONSTRUCTED_LENGTH=<number> - use <number> as length of constructed sequence of bytes for BER

The default setting is CODER_BER_CONSTRUCTED_LENGTH=1000.

- CODER_BER_CANONICAL_ON - restrict BER decode for SET type by canonical order.
- CODER_BER_CANONICAL_OFF - BER decode for SET type as defined in ITU X.690.

The default setting is CODER_BER_CANONICAL_ON.

Real values encoding

For encoding real values are divided into the following number values: (sign * 2^factor * number * base^exponent). It is the user option to choose which factor and which base should be used when encoding.

- CODER_BER_REAL_FACTOR_0 - use real factor 0 for coding
- CODER_BER_REAL_FACTOR_1 - use real factor 1 for coding
- CODER_BER_REAL_FACTOR_2 - use real factor 2 for coding
- CODER_BER_REAL_FACTOR_3 - use real factor 3 for coding
- CODER_REAL_BASE_2 - use real base 2 for encoding
- CODER_REAL_BASE_8 - use real base 8 for encoding
- CODER_REAL_BASE_16 - use real base 16 for encoding

The default setting is CODER_BER_REAL_FACTOR_0 and CODER_REAL_BASE_2.

Error checking configuration

- CODER_CHECK_NONE - remove all error checks from compiled library
- CODER_CHECK_NONE_VALUE - remove error checks of input values to encoding and decoding function from the compiled library

- CODER_CHECK_NONE_BUFFER - remove error checks in buffer management from the compiled library
- CODER_CHECK_NONE_DECODING - remove error checks in decoding functions from the compiled library
- CODER_CHECK_NONE_INNER - remove internal error checks in encoding and decoding functions from the compiled library

Note: Error checks in decoding functions

All decoding functions start with a check of type nodes and input values. These tests are not removed by defining the CODER_CHECK_NONE_DECODING macro. Checks in the buffer management are not affected by the macro either. You can remove these checks by using other macros in this list.

By default all checks are switched ON.

Encoding configuration

You can use the following switches to remove code for ASN.1 type from the compilation:

- CODER_NOUSE_BOOLEAN
- CODER_NOUSE_INTEGER
- CODER_NOUSE_NULL
- CODER_NOUSE_OBJECT_IDENTIFIER
- CODER_NOUSE_REAL
- CODER_NOUSE_BIT_STRING
- CODER_NOUSE_OCTET_STRING
- CODER_NOUSE_CHARACTER_STRING - NumericString, PrintableString, IA5String, VisibleString, UTCTime, GeneralizedTime
- CODER_NOUSE_ENUMERATED
- CODER_NOUSE_ENUMERTAED_ITEMS - ENUMERATED items if they are 0 1 2 3 etc

- CODER_NOUSE_SEQUENCE
- CODER_NOUSE_SET
- CODER_NOUSE_SET_OF
- CODER_NOUSE_SEQUENCE_OF
- CODER_NOUSE_CHOICE
- CODER_NOUSE_OPEN
- CODER_NOUSE_EXT - extension marker in ASN.1 sequence and set

You can use the following switches to redefine C type for ASN.1 type information field:

- CODER_ENUMERATED_TYPE=<type> - enumerated value. Default is “long”
- CODER_TagNumber_TYPE=<type> - ASN.1 tag class number. Default is “unsigned long”
- CODER_NumOf_TYPE=<type> - number of components. Default is “unsigned long”
- CODER_INTEGER_LBOUND_TYPE=<type> - low bound in integer constraint. Default is “long”
- CODER_INTEGER_UBOUND_TYPE=<type> - upper bound in integer constraint. Default is “long”
- CODER_SIZE_LBOUND_TYPE=<type> - low bound in size constraint. Default is “unsigned long”
- CODER_SIZE_UBOUND_TYPE=<type> - upper bound in size constraint. Default is "unsigned long"

These compile switches can be set automatically. `asn1util` generates `asn1_cfg.h` file for the `-c` option where the above compile switches are included after `#define` directives to minimize coder library code size automatically, see also “Configuration file generation” on page 703 in chapter 13, *The ASN.1 Utilities*. To be able to use this `asn1_cfg.h` automatic configuration you should define the following compile switch:

- CODER_AUTOMATIC_CONFIG - enables automatic configuration generated to the `asn1_cfg.h` file. This switch can be set by choosing the corresponding option in the Targeting Expert.

Buffers configuration

- CODER_BMS_SMALLBUF - uses small buffer implementation. The second argument of BufInitBuf interface - type of the buffer (see "tBMSBufType" on page 2851) will not be in use for this case.
- CODER_REMOVE_SMALLBUF - removes small buffer code from the compiled library

The default is including only small buffers to the library. When all buffers are absent, then only a null buffer will be available (a null buffer is a stub buffer without any functionality).

- CODER_USE_USERBUF - includes the user buffer into the library to be available at runtime
- CODER_BMS_USERBUF - uses the user buffer implementation if it is made available by previous switch
- CODER_BMS_TINY - uses a minimum set of bms interfaces. Interfaces to operate with buffer mode, character segment, bytes, bits also copy buffer interface are not available for this switch. It is allowed to use both buffer access types `tBuffer` and `tCoder` in this mode.

Example 487: Encoding in tiny mode ----------

```
tCoder coder;
tBMSUserMemory UserMemory;

BufInitBuf(&coder, bms_SmallBuffer);
ASN1_ENCODE(&coder,
            (tASN1TypeInfo *)&yASN1_Message,
            encValue);
BufCloseBufToMemory(&coder, &UserMemory);
...
BufInitBufWithMemory(&coder, &UserMemory);
ASN1_DECODE(&coder,
            (tASN1TypeInfo *)&yASN1_Message,
            decValue);
BufCloseBuf(&coder);
CUCF_FREE(UserMemory.MemPtr, UserMemory.MemSize, 0);
```

Example 488: `tBuffer` **access type in tiny mode**

```
tBuffer buf = NULL;
BufInitBuf(buf, bms_SmallBuffer);
```

> **Note:**
> The buffer value for this mode must be initialized by NULL before being used in buffer management. The behavior will otherwise be unpredictable.

Following buffer interfaces are not available in this mode:

```
BufGetMode, BufInNoMode, BufInReadMode, BufInWrite-
Mode

BufCopyBuf

BufInitWriteMode, BufCloseWriteMode

BufInitReadMode, BufCloseReadMode, BufCloseDele-
teReadMode

BufGetByte, BufPeekByte, BufPutByte

BufGetSeg, BufSkipSeg, BufPeekSeg, BufPutSeg

BufPutBit, BufGetBit, BufPutBits, BufGetBits
```

- CODER_SMALLBUF_SIZE=<number> - <number> defines small buffer allocation block

The default setting for the size is CODER_SMALLBUF_SIZE=0x1000.

Memory management configuration

- CODER_MMS_SDT - use SDT alloc/free procedures when working with memory in the coder library functions
- CODER_MMS_USER - apply user defined alloc/free functions when working with memory in the coder library functions (file “mms_user.h“ will be added for library compilation)

By default standard malloc and free functions from <stdlib.h> are used.

Error output configuration

- CODER_EO_SDT - use SDT error output functions (when this switch is turned on, SDT error output switch XECODER can influence error output functionality)

- CODER_EO_USER - apply user defined error output procedures to the library and use them for error output (file “ems_eo_user.h“ will be added for library compilation)
- CODER_EO_DEBUG - use internal library functions for printing error messages
- CODER_EO_NONE - do not output error messages text

The default setting is CODER_EO_SDT.

- CODER_TI_NAMES - enable name information in the Type Info.

Error handling configuration

- CODER_PATH_DEEP=<number> - nesting coefficient. It depends on the ASN.1 specification. The nesting coefficient is the maximum number of nested SEQUENCE, SET, CHOICE, SEQUENCE OF, SET OF or open types. By default `CODER_PATH_DEEP` is defined to 16. If the nesting coefficient in the ASN.1 specification is greater then 16 then you should define the `CODER_PATH_DEEP` compile switch with this value yourself, otherwise “segmentation fault” error will occur. The nesting coefficient for the following example is 4:

```
MySeq ::= SEQUENCE { -- NC=4
  a MyChoice,
  b NULL
}
MySet ::= SET { -- NC=1
  a INTEGER,
  b BOOLEAN
}
MyChoice ::= CHOICE { -- NC=3
  a MySeqOf,
  b MySetOf
}
MySeqOf ::= SEQUENCE OF MySet -- NC=2
MySetOf ::= SET OF BOOLEAN -- NC=1
```

- CODER_REMOVE_PATH - do not use detailed error messages (see “Detailed error messages” on page 2871) and user error handling (see “User defined Error Handling” on page 2880)

User data configuration

- CODER_USER_DATA - include `tUserData` defined in the `user_data.h` into the coder buffer, see “User Data” on page 2884

Value management configuration

- CODER_VMS_SDT - use SDT value interface when accessing external values from the encoding and decoding functions (this switch comes together with SDT switch XUSE_GENERIC_FUNC, which chooses generic value representation out of two available value representations for SDT/C values, by default it is turned off and old (backwards compatible) value representation is used)
- CODER_VMS_TTCN - use TTCN value interface when accessing external values from the encoding and decoding functions
- CODER_VMS_USER - choose user value interface for accessing external values from the encoding and decoding functions (file “vms_vr_user.h“ must be edited to configure user value access, it is included for library compilation)

By default CODER_VMS_SDT is chosen.

Printing configuration

- CODER_VMS_PRINT - enable debug printing opportunities (see “Printing Opportunities” on page 2885)

By default the coder library is compiled without debug printing functionality.

Generating Environment Files with Coding

You can call the encoding and decoding functions from the environment functions if you do not want to manipulate the encoded bit pattern explicitly in your SDL system.

The SDL Suite tools generate templates for the environment files that contains all necessary calls to buffer management functions, encoding functions and decoding functions. You can, of course, adapt this template to your needs or even write all the calls yourself. See chapter 57, *Building an Application* or chapter 65, *The Cmicro SDL to C Compiler*.

The template solution in the environment files has the following properties:

- One global buffer reference created at start-up of system. This buffer reference is used in all calls for all signals and for both encoding and decoding.

- All signals out from the system are encoded into bit-patterns. All parameters of the signals are encoded.
- All signals into the system decodes bit-patterns.

Compiling and Linking

The instructions you must perform in order to compile and link are listed in chapter 8, *Tutorial: Using ASN.1 Data Types, in the Getting Started*.

Chapter

59 *The Targeting Expert*

When doing targeting there is a lot of information to be provided, for instance how to generate the C source code, how to configure the SDL to C compiler, which compiler to be used etc.

The Targeting Expert is a tool managing the complete process of targeting (for the Cadvanced, Cmicro and Cextreme SDL to C compilers only).

The easiest way to get an executable from an SDL specification (just generate C code for a complete SDL system - compile and link the executable) is as well supported as more complex cases. Therefore there are several Pre-defined Integration Settings which are ready-to-use.

Furthermore the Targeting Expert is able to re-use the deployments done in the Deployment Editor (see "The Deployment Editor" on page 1769 in chapter 40, *The Deployment Editor*) and to configure any single option thinkable in the process of doing targeting with SDL Suite. Those options can be the SDL to C compiler's options, the compiler, linker and make tool options, for example.

After all this chapter is a description of the Targeting Expert Interactive Mode and a reference for the Batch Mode.

A list of FAQs (frequently asked questions) can be found at the end of this document.

Introduction

The Targeting Expert assists you in setting up and configuring a complete target application. A sub-set of the Targeting Expert functionality can be used in Batch Mode when it is only desired to re-build the complete system (or just several components).

Starting the Targeting Expert

In graphical Mode

Starting the Targeting Expert in the graphical mode means that the user interface is visible and you can directly interact by selecting function and menu items.

You enter the graphical mode by selecting *Targeting Expert* in the Organizer's *Generate* menu or by using the following command line options:

sdttaex [-h | -v | -pdm `<partition_diagram>.pdm` | -sdt `<systemname>.sdt` | -pr `<systemname>.pr`] [-t `<target>`] [-hostsim | -realsim | -sim | -val | -makeall]

Option	Relevance
-v	The version number is shown and the application exits.
-h	Information about the usage is shown and the application exits.
-pdm `<partition_diagram>.pdm`	The partitioning diagram file to be used.
-sdt `<systemname>.sdt`	The system file to be used.
-pr `<systemname>.pr`	The system's PR file to be used
-t `<target>`	The qualifier of the SDL system's part to build
-makeint `<integration>`	Make the given integration
-makeall	Re-make last configuration

More information about the user interface can be found in "The Main Window" on page 2904.

See "Interactive Mode" on page 2910 to get familiar with the Targeting Expert functionality.

In Batch Mode

The command line mode is also called Batch Mode. Please see the appropriate sections for a more detailed description.

Possible command line options are:

taexbatch [-v | -h | -t `<target>`| -yes | -no]

Option	Relevance
-v	The version number is printed to stdout and the application exits.
-h	Information about the usage is printed to stdout and the application exits.
-t `<target>`	The qualifier of the SDL system's part to build
-yes	All the questions that probably come up will be answered with 'yes'.
-no	All the questions that probably come up will be answered with 'no'.

For more information about the commands allowed in a batch file see "Description of Batch Mode Commands" on page 2962.

Note:

Targeting Expert does not startup properly if there is more than one instance of the Postmaster running (sdt.exe on Windows and sdt on UNIX). Be sure you have at most one SDL Suite running when starting Targeting Expert.

The Graphical User Interface

This section describes the appearance and functionality of the graphical user interface of the Targeting Expert (sdttaex). Some user interface descriptions common to all tools can be found in chapter 1, *User Interface and Basic Operations*.

The Main Window

When you start the Targeting Expert in the interactive mode its main window is displayed.

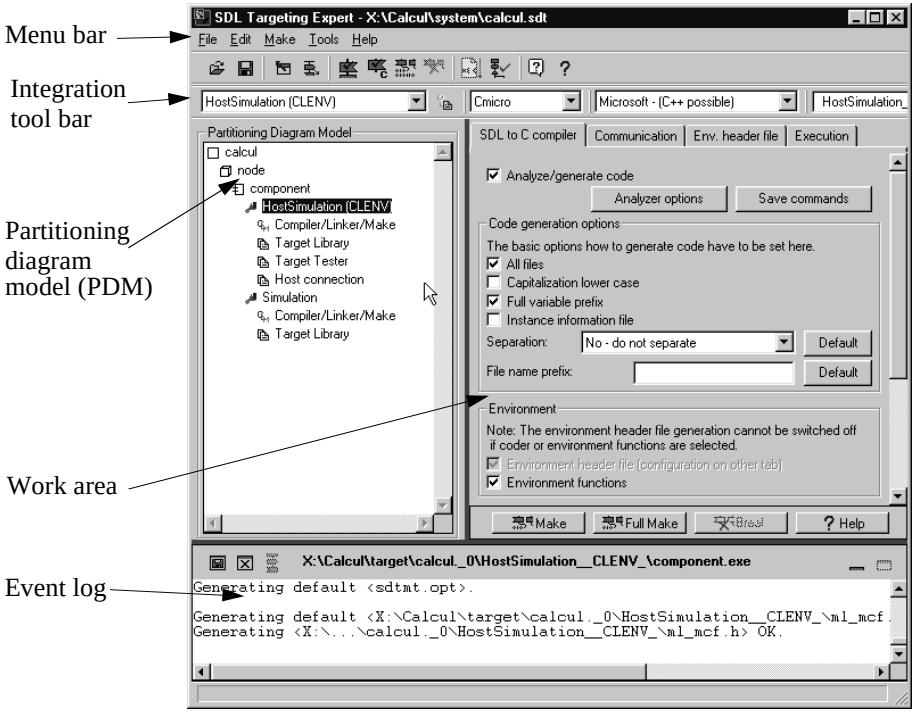


Figure 515: The Targeting Expert main window

The Menu Bar

This section describes the menu bar of the Targeting Expert Main window and all the available menu choices.

The menu bar contains the following menus:

- *File Menu*
- *Edit Menu*
- *Make Menu*
- *Tools Menu*
- *Help Menu*
  (See "Help Menu" on page 15 in chapter 1, *User Interface and Basic Operations*.)

Configurable Menus

Some menu choices may be available through the concept of user-defined menus. For more information, see "Defining Menus in the SDL Suite" on page 18 in chapter 1, *User Interface and Basic Operations*.

The definition file for user-defined menus searched for by the Targeting Expert is called `taex-menus.ini`.

There are several placeholders possible to be used with the `FormattedCommand` clause in `taex-menus.ini`.

placeholder	... will be replaced by ...
%s	system directory
%b	target directory (the one given in the Organizer)
%t	sub target directory (the one calculated depending on the selected component and integration)
%e	executable name (inclusive extension)
%i	intermediate directory

File Menu

The *File* menu contains the following menu choices:

- *Open*
- *Save*

- *Exit*

See "File Menu" on page 8 in chapter 1, *User Interface and Basic Operations*.

Edit Menu

The *Edit* menu gives access to the enlargement of the Targeting Expert configuration files.

- *Add Compiler*
- *Edit Compiler Section*
- *Remove Compiler*
- *Add Communications Link*
- *Remove Communications Link*
- *Edit Makefile*

  A text editor is displayed where you can modify the generated makefile.

- *Edit Configuration Header File*

  A text editor is displayed where you can modify the generated configuration header file.

- *Edit Environment File*

  A text editor is displayed where you can modify the generated environment C file.

Make Menu

The *Make* menu allows to control the make process of the Targeting Expert.

- *Analyze*

  The selected integration will be analyzed.

- *Generate code*

  C code will be generated for the selected integration.

- *Make all*

  All the configured components will be made again.

- *Clean*

All the object files in the object directory will be removed.

Tools Menu

The *Tools* menu gives access to other SDL Suite tools.

- *Show Organizer*

  The Organizer main window is displayed.

- *Load SDL System*

  The partitioned SDL system that is currently worked on in the Targeting Expert can be loaded into the Organizer. Use this entry if the Targeting Expert main window is displayed after the execution in the batch mode has failed.

- *Utilities > DOS to UNIX*

  It is possible to specify a list of ASCII files. All the files in this list will be converted from DOS to UNIX style. See "DOS to UNIX" on page 3010

- *Utilities > UNIX to DOS*

  It is possible to specify a list of ASCII files. All the files in this list will be converted from UNIX to DOS style. See "UNIX to DOS" on page 3010

- *Utilities > Indent*

  The indentation of all the ASCII files in the list of files specified by the user will be corrected. See "Indent" on page 3010

- *Utilities > Preprocess*

  The list of generated C files that is specified by the user will be preprocesses. See "Preprocessor" on page 3011

- *Wizards > TCP/IP communication*

  A wizard dialog pops up and allows you to configure the TCP/IP communication between different components. See "TCP/IP signal sending" on page 2952

- *SDL > Simulator UI*

This menu choice starts a new, empty Simulator UI. Several Simulator UIs may exist at the same time. See *"Graphical User Interface" on page 2199 in chapter 49, The SDL Simulator*.

- *SDL > Explorer UI*

  This menu choice starts a new, empty Explorer UI. Several Explorer UIs may exist at the same time. See *"Graphical User Interface" on page 2349 in chapter 52, The SDL Explorer*.

- *SDL > SDL Target Tester*

  This menu choice starts a new, empty SDL Target Tester UI. See "Graphical User Interface" on page 3656.

- *Customize*

  A dialog pops up and allows to customize the Targeting Expert. See "Customization" on page 2924 for more details.

The Integration Tool Bar

The integration tool bar should be used to (from left to right)

- Select the Pre-defined Integration Settings
- Get help about the selected integration
- Select an SDL to C Compiler
- Select a C Compiler to be used

Furthermore it should also be used to handle pre-defined and user-defined integration settings (see "Handling of Settings" on page 2920), i.e.

- to set a new user settings file name
- to export user settings as pre-defined settings
- to import node or application settings

The Work Area

The work area of the Targeting Expert is used for giving input masks for the different configurations and scalings.

The different input masks are described in

- "Configure how to Make the Component" on page 2948
- "Configure Compiler, Linker and Make" on page 2930
- "Configure and Scale the Target Library" on page 2946

- "Configure the SDL Target Tester (Cmicro only)" on page 2947
- "Configure the Host (Cmicro only)" on page 2948

The Event Log

The Targeting Expert has an event log containing information about found inconsistencies, read/write problems concerning file access, output, etc.

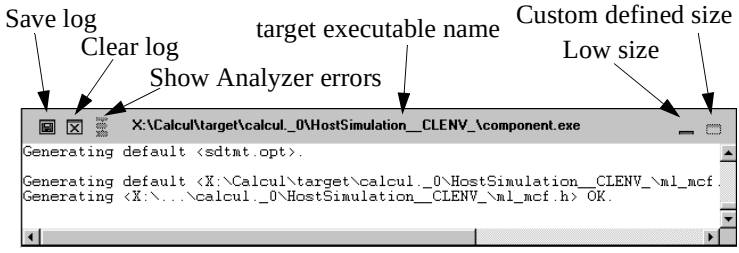


Figure 516: The event log

Analyzer/Compiler Errors

You can access analyzer and compiler errors by double clicking the error message in the event log.

- In case of an analyzer error the SDL Editor is displayed with the erroneous SDL symbol highlighted.
- In case of a compiler error the selected editor is displayed with the erroneous line highlighted.
  Note 1: the selected editor must be able to highlight the desired line. It has to be selected in the Customization of the Targeting Expert.
  Note 2: the compiler's error message syntax must be described in the Compiler Error Descriptions section in file `sdttaex.par` for the compiler used.

Interactive Mode

You can use the Targeting Expert interactive mode to achieve different targets to:

- Build an un-configured or optimized target executable by following the steps described in "Targeting Work Flow" on page 2926.
- Configure the distributed SDL Suite release and handle the already done settings. Please see below.

Although most of the steps in targeting are supported in the Targeting Work Flow you sometimes need access to other functionality, for example:

- Compiler Definition for Compilation to the target library's known compilers.
- Communications Link Definition for Compilation (for the inter-communication with the SDL Target Tester's host application).
- Handling of Settings
- Customization

Compiler Definition for Compilation

> **Note:**
>
> All the modifications that can be done here are only valid for the current system, i.e. all the information will be stored into the system's target directory.

Add

The following is applicable only if using the Cmicro SDL to C compiler:

Select *Add a new Compiler* from the *Edit* menu. The Add compiler dialog is displayed.

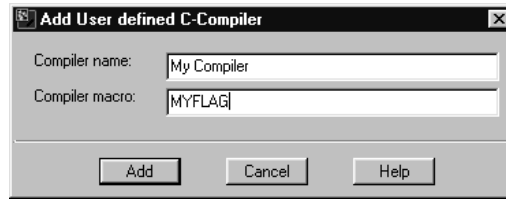


Figure 517: Add compiler dialog

- Compiler Name

  The text you enter here will be used to identify the compiler in the *Compiler Flag* list.

- Compiler Macro

  The macro name you enter here will be used to identify the compiler in the target library. The compiler macro must fit to `[A-Za-z_][A-Za-z0-9_]*`

When you click *Add* the compiler is added to the `c*_conf.def` file. Please see "Configuration Files" on page 2974 for information about the file's syntax and duty.

Edit

For each compiler that is supported by a target library there is a specific section in

- `scttypes.h` (Cadvanced), see "Compiler Definition Section in scttypes.h" on page 3147.
- `ml_typ.h` (Cmicro), see "Adaptation to Compilers" on page 3523 in chapter 66, *The Cmicro Library*
- `extreme_kern.h` (Cextreme), see Compile and Link an Application.

Whenever there is a compiler flag defined, not known by the library, a file called `user_cc.h` (for Cadvanced and Cmicro) or `comphdef.h` (for Cextreme) will be included which has to contain the compiler specific settings.

The dialogs shown below are input masks that request all the needed information to generate such a file. Select *Edit Compiler Section* from the Edit Menu.

Cadvanced

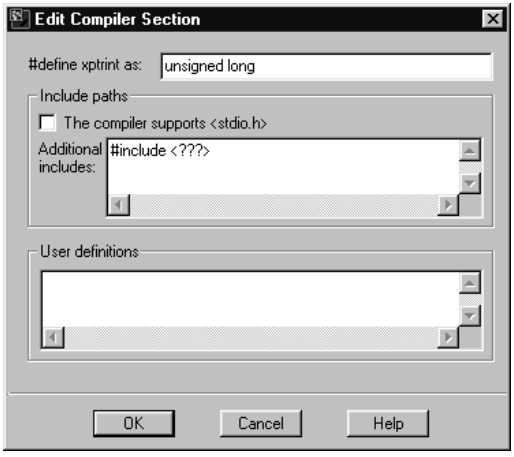


Figure 518: Edit compiler section dialog (Cadvanced)

For Cadvanced the following is requested:

- *#define xptrint*

  Enter the mapping for xptrint values; usually `unsigned long`.

- *Include paths*

  – *The compiler supports <stdio.h>*

  If selected this section will be generated:

  ```
  #ifdef XREADANDWRITEF
    #include <stdio.h>
  #endif
  ```

  Otherwise nothing will be generated here.

  – *Additional includes*

  The contents of the edit box will simply be copied in to the `user_cc.h` file. It should carry something like `#include <file>`.

- *User definitions*

  Enter user definitions for the `user_cc.h` file.

Cmicro

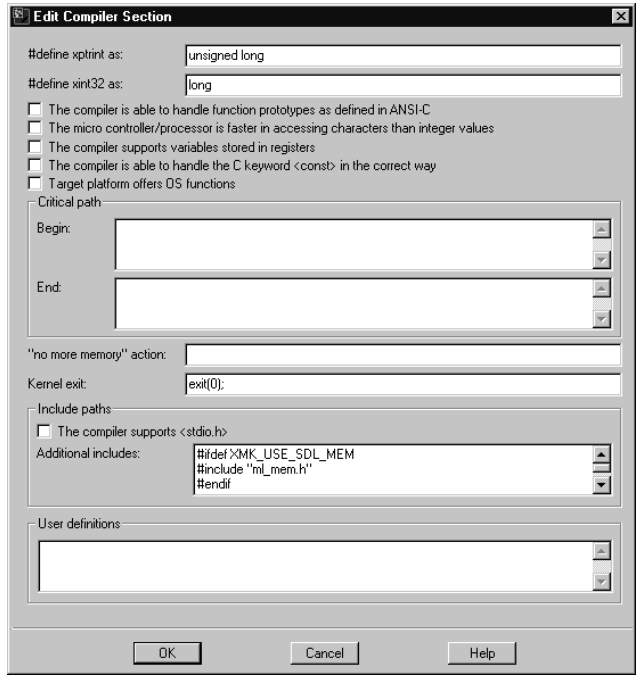


Figure 519: Edit compiler section dialog (Cmicro)

For Cmicro the following is requested:

- *#define xptrint*

  Enter the mapping for `xptrint` values; usually `unsigned long`.

- *#define xint32*

  Enter the mapping for `xint32` values; usually `long`.

- *The compiler is able to handle function prototypes as in ANSI C*

  – If selected this section will be generated:

```
#undef XNOPROTO
#define XPP(x) x
#define PROTO(x) x
```

– If **not** selected this section will be generated:

```
#define XNOPROTO
#define XPP(x)
#define PROTO(x)
```

- *The micro controller/processor is faster in accessing characters than integer values.*

  – If selected this section will be generated:

  ```
  #define xmk_OPT_INT char
  ```

  – If **not** selected this section will be generated:

  ```
  #define xmk_OPT_INT integer
  ```

- *The compiler supports variables stored in registers*

  – If selected this section will be generated:

  ```
  #undef X_REGISTER
  #define X_REGISTER register
  ```

  – If **not** selected this section will be generated:

  ```
  #undef X_REGISTER
  #define X_REGISTER
  ```

- *The compiler is able to handle the C keyword <const> in the correct way*

  – If selected this section will be generated:

  ```
  #undef XCONST
  #define XCONST const
  ```

  – If **not** selected this section will be generated:

  ```
  #undef XCONST
  #define XCONST
  ```

- *Target platform offers OS functions*

  – If selected this section will be generated:

  ```
  #define XMK_USE_OS_ENVIRONMENT
  ```

  – If **not** selected this section will be generated:

  ```
  #undef XMK_USE_OS_ENVIRONMENT
  ```

- *Critical paths*

  Enter the commands which are needed to have critical paths, i.e. how to disable and enable interrupts.

Note:

All the commands you enter will be copied in to `user_cc.h`. If there is more than one line of source code, each line must end with a '\' because the compiler's preprocessor depends on it.

- *"no more memory" action*

  Enter the action (e.g. a function call) to process if no more memory can be allocated.

- *Kernel exit*

  Enter the function call to process if the kernel should be exited. DEfault is `exit();`

- *Include paths*

  – *The compiler supports <stdio.h>*

  If selected this section will be generated:

```
#ifdef XMK_ADD_PRINTF
  #include <stdio.h>
#endif
```

  Otherwise nothing will be generated here.

  – *Additional includes*

  The contents of the edit box will simply be copied into the `user_cc.h` file. It should carry something like `#include <file>`.

- *User definitions*

  Enter user definitions for the `user_cc.h` file.

Remove

The following is applicable only if using the Cmicro SDL to C compiler:

To remove a compiler from the private `c*_conf.def` file, select *Remove an Unused Compiler* from the Edit Menu

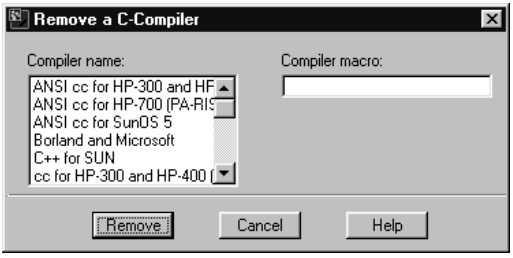


Figure 520: Remove compiler dialog

Note:

The file `user_cc.h` will not be affected by removing a compiler from the `c*_conf.def` file.

Communications Link Definition for Compilation

The following is only applicable if using the Cmicro SDL to C compiler.

Note:

All the modifications that can be done here are only valid for the current system, i.e. all the information will be stored into the system's target directory.

Add

To allow the selection of a new communications link in the Targeting Expert user interface the appropriate macros should be given in the Targeting Expert.

Select *Add a User Defined Communications Link* from the Edit Menu.

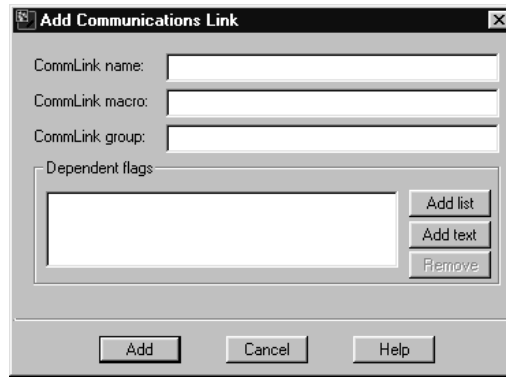


Figure 521: Add a user defined communications link

- *CommLink Name*

  The text will be shown in the Targeting Expert UI to identify it in the dialog described in "Configure the SDL Target Tester (Cmicro only)" on page 2947.

- *CommLink Macro*

  The macro name will be used to identify the communications link in the target library. The communications link macro must fit to `[A-Za-z_][A-Za-z0-9_]*`

- *CommLink Group*

  The text will be used to build a group of the communications link description done here, i.e. the CommLink macro and the dependent flags will be summarized as the group given here.

- *Dependent Flags*

  – *Add List*

    A dialog where you enter a Value List is displayed.

  – *Add Text*

    A dialog where you enter a Text Value Flag is displayed.

  – *Remove*

    The entry you have selected in the list box will be removed.

When you click *Add* the communications link is added to the `c*_conf.def` file. Please see "Configuration Files" on page 2974 for information about the file's syntax and duty.

Note:

The communications link source code has to be set up by the user.

Value List

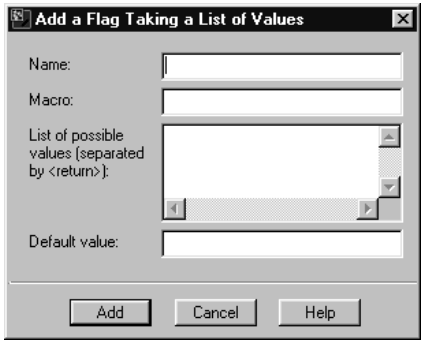


Figure 522: Add a flag taking a list of values

- *Name*

  The name is used for identification purposes in the Targeting Expert user interface.

- *Macro*

  The macro will be used for scaling purposes in the target library. It must fit to `[A-Za-z_][A-Za-z0-9_]*`

- *List of possible values*

  Enter a list of all allowed values the macro can take. Each entry has to be delimited by `<return>`.

- *Default value*

  Enter the default value which must be one of the possible values entered above.

When you click *Add* this value list is added to the communications link you entered in the dialog in Figure 521 on page 2917.

Text Value Flag

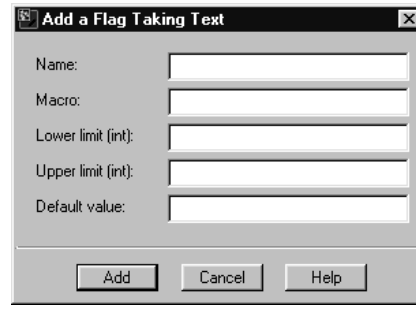


Figure 523: Add a flag taking text

- *Name*

  The name is used for identification purposes in the Targeting Expert user interface.

- *Macro*

  The macro will be used for scaling purposes in the target library. It must fit to `[A-Za-z_][A-Za-z0-9_]*`

- *Lower limit*

  Enter the lower limit if the value given with this macro is an integer value.

- *Upper limit*

  Enter the upper limit if the value given with this macro is an integer value.

- *Default value*

  The default value of the macro in this edit line has to be between the lower and upper limit if it is an integer value. Of course, it is possible to enter an alphanumeric value here, too.

When you click *Add* this value list is added to the communications link you entered in the dialog shown in Figure 521 on page 2917.

Remove

To remove a communications link from the private `c_conf.def` file, select *Remove a communications link* from the Edit Menu.

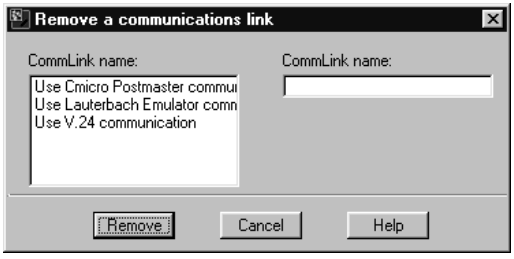


Figure 524: Remove a communications link

Handling of Settings

General

The integration tool bar on the main window offers four buttons to handle the settings, i.e. the way settings are stored and where they are taken from can be influenced.

Figure 525: Integration tool bar in the main window

Note:

The settings addressed in this section are only compiler, linker and make settings. The configuration of the SDL to C compiler library is not touched!

After selecting pre-defined integration settings the Targeting Expert automatically creates a file called `<integration_name>.uis` in the according component directory. (Please see "Target Sub-Directory Structure" on page 2996).

For some reasons it is sometimes useful to modify this way of saving the user done settings. E.g.

- To be able to “play” with settings it can be useful to switch to another file while the default one is not touched. After playing it is easy to switch back to the old settings by selecting the old file again.

- After doing an adaptation for a specific target the manually modified settings can be exported as new pre-defined integration settings. Please see “Export” on page 2922.

- Estimated there are several components inside of a node and each of them should be build the same way, then it is probably useful to do the adaptation of settings once on node level and to import these settings for each component.
  This also means that the compiler, linker and make settings can no longer be modified on component level.

- Estimated there are several components in different nodes that should be build the same way, then it is possible to import the setting from the application.
  This also means that the compiler, linker and make settings can not be modified any longer on component level.

Estimated there is an integration ’A’ and there are modified settings on this integration on application, node and component level. Then there are three different files `A.uis` saved in the correspondent application, node and component directories.
Figure 526 shows where the settings can be done and where they are taken from when the component is selected.

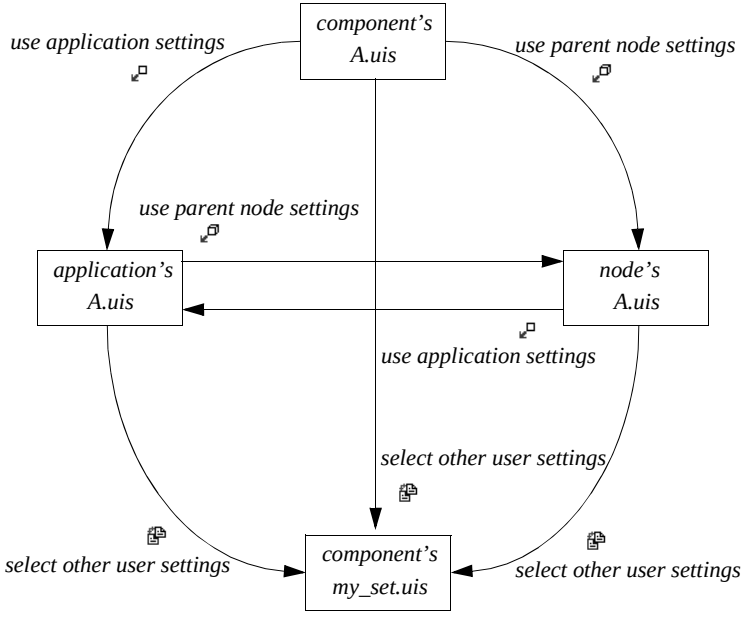


Figure 526: handling the user settings

Export

The settings of the currently selected integration will be exported as new Pre-defined Integration Settings with the name given in the first dialog that pops up.

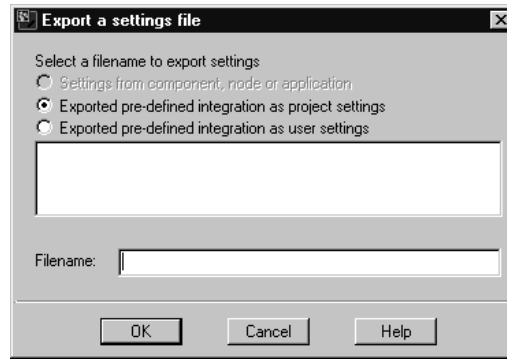


Figure 527

Note:

All the integrations must have unique names, i.e. it is not possible to export settings with an already known name.

- *Export pre-defined integration as project settings*

  In this case the new `.its` file is stored in the same directory as the `<systemname>.sdt` file. This has the advantage that all users working on the system can access the pre-defined integration.

- *Export pre-defined integration as user settings*

  The `.its` file is stored in the `<installationdir>` **in Windows** or respectively the `§HOME/.telelogic` directory **on UNIX**. This has the advantage that other users do not see all the new pre-defined integration settings.

In the second dialog a list of template files which have to be copied into the target directory can be specified. As default, the list of files from the "parent" settings will be given. Modify the list to your needs. Directly after exporting the settings they can be used for further configurations.

Note:

The sections Configuration Settings to Be Set and Configuration Settings to Be Reset are not exported.

Customization

The Targeting Expert interactive mode can be customized. After selecting the *Customize* entry in the Tools Menu the dialog shown in Figure 528 pops up.

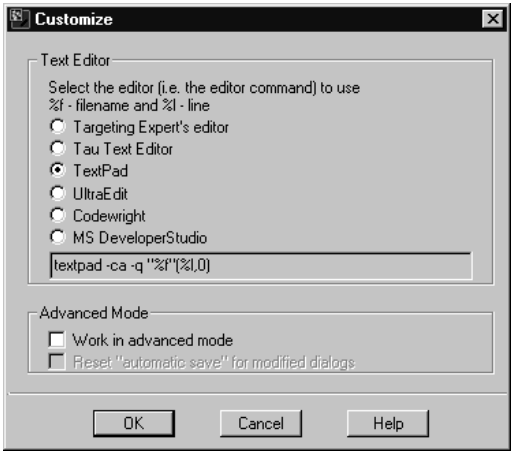


Figure 528: Customize dialog

Text Editor

The text editor which should be opened to edit or view text files can be selected here. The list of supported editors depends on the platform the Targeting Expert is running on (Windows/UNIX). It can be modified or extended in the file `sdttaex.par`. Please see "Parameter File sdttaex.par" on page 3000.

Advanced Mode

The advanced mode allows you to:

- do more configurations than possible in the non-advanced mode. See "More Configurations" on page 2925 for more information.
- switch between different dialogs without using the *Cancel* or *Save* button
- start the code generation automatically when selecting a new integration

- configure all compilers. Even the ones that are not available on the used platform.

More Configurations

- Normally there is no need to modify the compiler flag if using a pre-defined integration, but in the advanced mode the Compiler Flag can be modified.
- The standard pre-defined integrations do not allow you to use all the configuration options in the Configure and Scale the Target Library dialog (disabled). If the advanced mode is switched on, it is possible to switch on and off all the options supported by the SDL to C compiler library.

Caution!

The pre-defined integrations distributed are tested only with the given library configuration. It is not guaranteed that it will work for all kinds of modifications!

Targeting Work Flow

Introduction

You can start the Targeting Expert form the Organizer's *Generate* menu when having a deployment diagram, the SDL system or a block/process of the SDL system selected.

Estimated the SDL system is selected the Targeting Expert converts the system into a default partitioning diagram model (deployment diagram). This is done because the Targeting Expert can only handle partitioning diagram models as an input. The tree window shown in Figure 529 gives an idea how the default partitioning diagram model looks like.

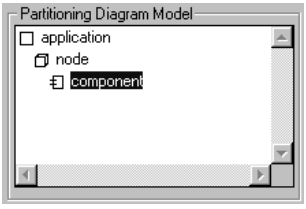


Figure 529: Partitioning diagram in the Targeting Expert

The different entries have got the following meaning:

	application This is the collection of all the deployed SDL systems. It is used as an object to allow configurations of several nodes only, i.e. it is not possible to make an application.
	node A node symbolizes an instance of the platform/computational resource that components execute on, i.e. it is not possible to make a node.
	component A component is interpreted as an executable program.

I.e. an application or node can be configured but not build. Only components can end up in executable programs.

In the start-up phase, directly after the partitioning diagram model has been displayed, the Targeting Expert generates a sub-directory structure into which all the configuration settings, object files and so on will be put. Please see "Target Sub-Directory Structure" on page 2996 for more information.

Now you can use the Targeting Expert to configure each component.

> **Hint:**
>
> If you switch the Targeting Assistant on in the "Help" menu, there are tool tips displayed for each entry on the main window.

Operation Steps

The following operation steps should be done at least once when doing targeting for a component the very first time. When the Targeting Expert is used again later to optimize the target it is of course not necessary to do all the steps once more.

1. Actions to perform:
   - Select the component which should be configured.

> **Note:**
>
> All the settings you define and the actions automatically performed by the Targeting Expert have an influence only on the component selected in the Targeting Expert tree window (see Figure 529 on page 2926).
>
> If the configuration should be re-used (this is sometimes reasonable if several components should be built using exactly the same settings), you can modify the way your settings are handled. Please see "Handling of Settings" on page 2920.

2.

Actions to perform:

- Select the Pre-defined Integration Settings
- Select an SDL to C Compiler (if not already done in the pre-defined integration settings)
- Select a C Compiler (if not already done in the pre-defined integration settings)

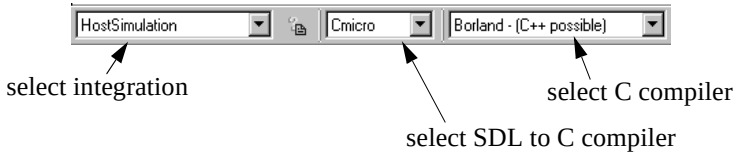


Figure 530: Integration selection in the main window

Select the Pre-defined Integration Settings

With every SDL Suite distribution there are several Pre-defined Integration Settings which you can use to get a target executable in an easy manner.

To get an optimized target executable concerning size and speed these pre-defined integration settings should only be seen as templates.

Please see "Distributed Pre-defined Integration Settings" on page 2979.

The pre-defined integration settings can be selected in the integration tool bar in the main window (see Figure 530). All the pre-defined integration settings are sorted into the groups

- Simulations
- Validations
- Target Tests
- Light Integrations
- Threaded Integrations

Additionally the integration `<user defined>`, which does not belong to an integration type and is not pre-defined, can be selected.

Hint:

If there is no set of pre-defined integration settings that fits your needs (e.g. a not yet supported compiler), it is recommended to select `<user-defined>` and to do the further configurations by hand.

Select an SDL to C Compiler

The SDL to C compiler to be used will automatically be set if you select one of the pre-defined integration settings. It is only necessary to select the SDL to C compiler if the `<user-defined>` settings have been selected. All the available SDL to C compilers will be given in the combo box shown in Figure 530.

Depending on the licenses found the following SDL to C compilers are supported:

- Cadvanced
- Cmicro
- Cextreme

Note:

Directly after the SDL to C compiler is selected (i.e. after pre-defined integration settings are selected) the Targeting Expert checks if there is already an automatic configuration file `sdl_cfg.h` or `extreme_user_cfg.h` (done by the SDL to C compiler) available.

If not, the SDL to C compiler should be invoked to generate an automatic configuration.

For Cadvanced this is only done if the flag `USER_CONFIG` is set in the compiler options (e.g. `-DUSER_CONFIG`).

Select a C Compiler

After a pre-defined integration setting is selected the Targeting Expert tries to automatically set the compiler to be used, i.e.

- if there is only one compiler supported with the selected pre-defined integration settings than that one is set.
- if there are more than one compiler supported the Targeting Expert tries to set the default compiler selected in the Preferences. (Please see chapter 3, *The Preference Manager*)

However, it is possible to set another compiler by selecting it in the combo box shown in Figure 530.

If the *<user defined>* integration has been chosen, a new compiler can be added by selecting the entry “Add new compiler description”.

3. Actions to perform:

- Configure Compiler, Linker and Make
- Configure and Scale the Target Library
- Configure the SDL Target Tester (Cmicro only)
- Configure the Host (Cmicro only)

To select the desired configuration dialog, please select the correspondent entry in the partitioning diagram.

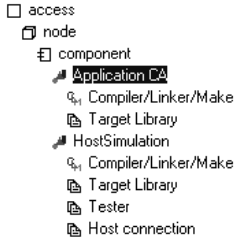


Figure 531: Tree structure in the partitioning diagram

Configure Compiler, Linker and Make

This part of the configuration is divided into four sub-steps which are taken from the selected pre-defined integration settings as far as possible.

Compiler

You enter the needed compiler configuration in a special input mask. The compiler you specify here will be used to compile the generated code and the target library.

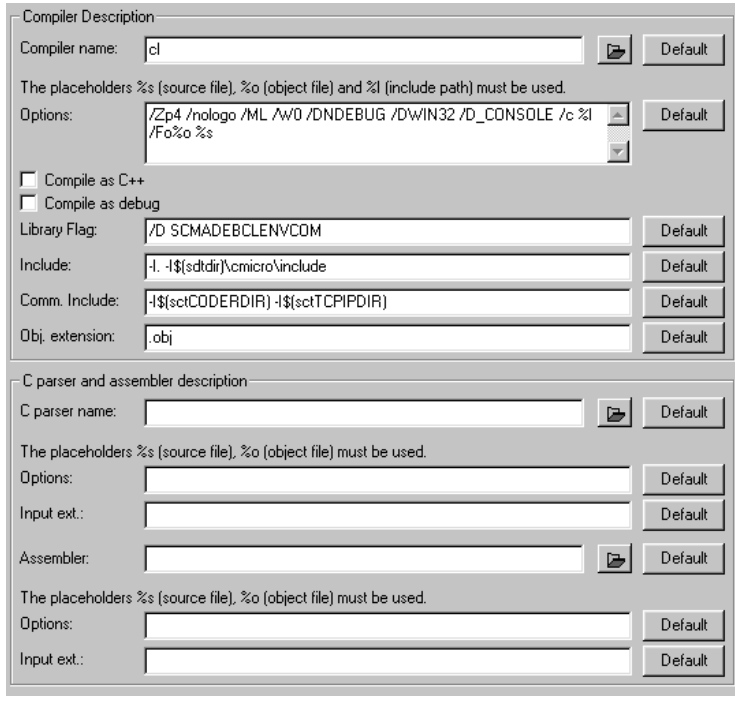


Figure 532: Compiler configuration

Click *Default* if you want to restore the default values.

- *Compiler Name*

  Enter the name of the compiler application.

- *Options*

Enter the options given to the compiler as command line arguments here.

- Enter the placeholder `%s` where the source file name of the file to be compiled has to be inserted (used for the makefile generation).
- Enter the dummy parameter `%o` where the object file's name has to be inserted (used for the makefile generation).
- Enter the dummy parameter `%I` where the include path option has to be placed (see below).

- *Compile as C++*

  The compiler options to compile C files as C++ files will be added/removed from the Options. For a definition of the used compiler options see the "Parameter File sdttaex.par" on page 3000.

- *Compile as debug*

  The compiler options to generate the object files including debug information will be added/removed from the Options. For a definition of the used compiler options see the "Parameter File sdttaex.par" on page 3000.

- *Library Flag*

  The compiler option to define a flag plus the C macro used to select the desired library must be given here. (When using Cmicro this field is empty per default.)

- *Include*

  Enter the compiler option needed to specify include paths and the include paths themselves here. The complete contents of this entry will replace the entry `%I` in the Options (see above).

  The include path can contain the following environment variables (The environment variables will be expanded during the make execution):

Variable	Path it points to
`sdtdir`	`<installationdir>/sdt/sdtdir/<platform>` as long as Library directory is not set

Variable	Path it points to
`scttargetdir`	Absolute path to the current target directory (depends on the selected component)
`sctobjdir`	`$(scttargetdir)` + the relative path selected in Object directory
`sctuseinclude` (Cadvanced only)	`$(sdtdir)/INCLUDE`
`sctincludedir` (Cmicro only)	`$(sdtdir)/cmicro/include`
`sctkerneldir` (Cmicro only)	`$(sdtdir)/cmicro/kernel`
`scttesterdir` (Cmicro only)	`$(sdtdir)/cmicro/tester`

- *Comm Include*

  The compiler option needed to specify include paths and the include paths to the coder library must be given here. (Only used if the ASN.1 and/or SDL coder functions are generated and used)

- *Obj. extension*

  Enter the object extension to be used for the compiler output.

If only the information up to here is specified in this dialog, each C file is compiled in one single step as shown in Figure 533.

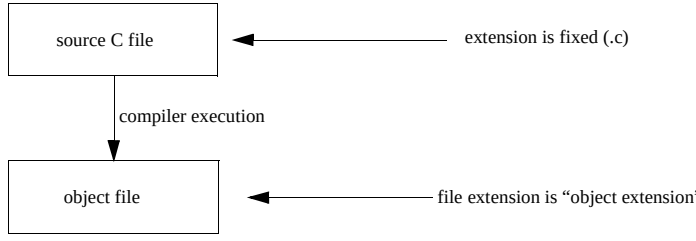


Figure 533:Single step compilation model

For some target compilers, compilation is done in three steps using three different tools. If that is the case, the compilation of each C file is

done like shown in Figure 534. In this case the following entries also need to be given.

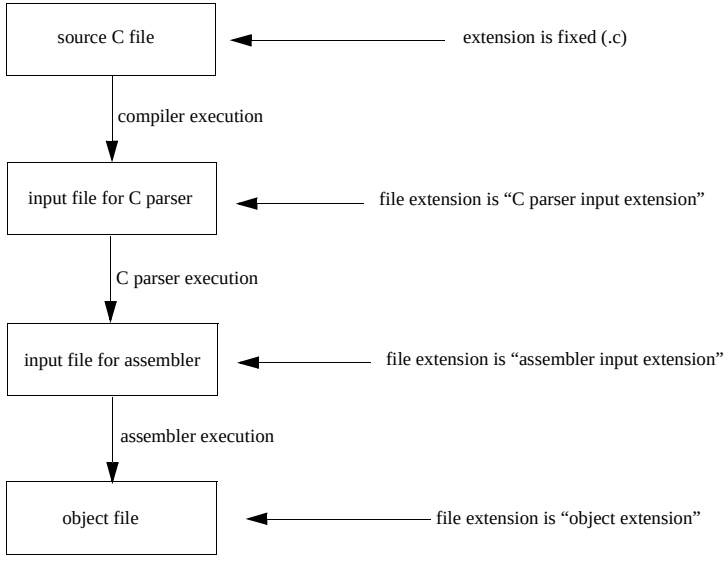


Figure 534:Three step compilation model

- *C parser name*

  The name of the C parser application has to be given here.

- *Options (C parser)*

  Enter the options given to the C parser as command line arguments here.

  – Enter the placeholder `%s` where the input file name of the file to be parsed has to be inserted (used for the makefile generation).
  – Enter the dummy parameter `%o` where the output file’s name has to be inserted (used for the makefile generation).

- *Input extension (C parser)*

  Enter the input file’s extension for the C parser.

- *Assembler*

The third step to process is the execution of the assembler. Enter the name of the assembler application here.

- *Options (Assembler)*

  Enter the options given to the assembler as command line arguments here.

  – Enter the placeholder `%s` where the input file name of the file to be assemble has to be inserted (used for the makefile generation).
  – Enter the dummy parameter `%o` where the output file's name has to be inserted (used for the makefile generation).

- *Input Extension (Assembler)*

  The input file's extension used by the assembler has to be specified in this edit line.

Source Files

All the files (except the coder and the generated files) which will be compiled and linked to the target executable are listed here.

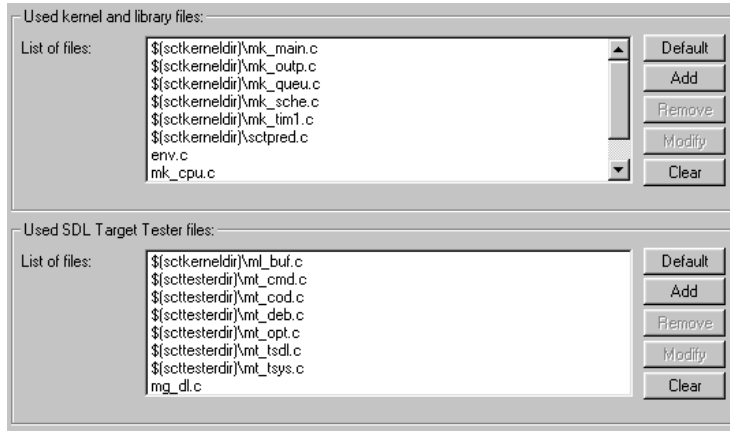


Figure 535: Source files used

- *Used kernel and library files* and
  Used SDL Target Tester files (Cmicro only)

  – *Default*

    Restores the default values.

  – *Add*

    A file selection dialog is opened where you can select the file which should be added to the list of kernel files.

  – *Remove*

    Removes the item you have selected in the list box.

  – *Modify*

    A dialog pops up and the selected entry can be modified.

  – *Clear*

    Deletes all the items in the list box.

Compiler Flag

> **Note:**
> This section is only available if the Advanced Mode is selected and the SDL to C compiler Cmicro is used.

When compiling the generated code and/or the SDL to C compiler's library you need to set a compiler flag to adjust the code.

The list of available compilers shown in Figure 536 belongs to the Cadvanced target library.

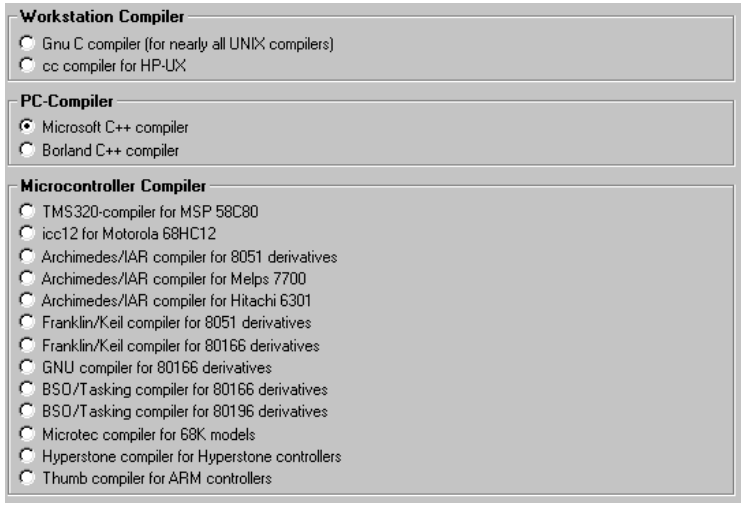


Figure 536: Flag selection

Caution!

You must select the correct compiler flag in order to avoid compilation errors.

For more information concerning the compiler specific adjustment of the libraries see

- `scttypes.h` (Cadvanced), see "Compiler Definition Section in scttypes.h" on page 3147.
- `ml_typ.h` (Cmicro), see "Adaptation to Compilers" on page 3523 in chapter 66, *The Cmicro Library*
- `extreme_kern.h` (Cextreme), see Integration with Compiler and Operating System.

Additional Compiler

If there are other files than the generated ones or the SDL to C compiler's library to be compiled all the requested things in this section have to be entered.

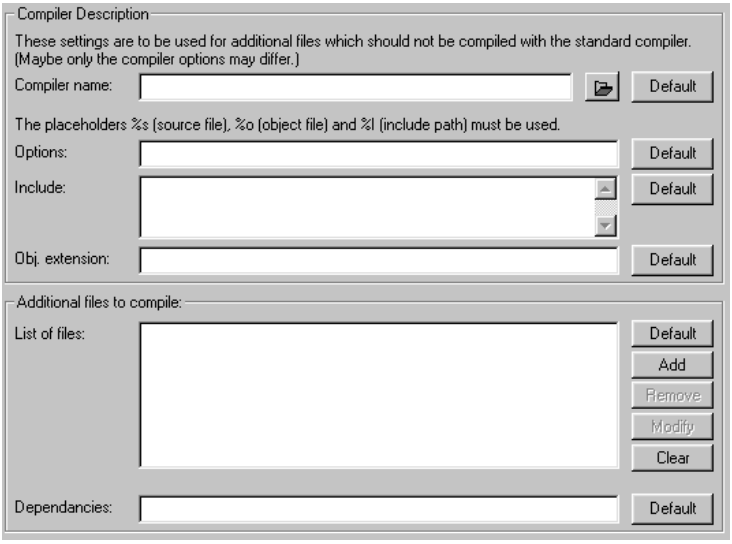


Figure 537: Additional compiler configuration

- *Compiler Name*

  Enter the name of the compiler application.

- *Options*

  Enter the options given to the compiler as command line arguments here.

  – Enter the placeholder `%s` where the source file name of the file to be compiled has to be inserted (used for the makefile generation).
  – Enter the dummy parameter `%o` where the object file's name has to be inserted (used for the makefile generation).
  – Enter the dummy parameter `%I` where the include path option has to be placed (see below).

- *Include*

  Enter the compiler option needed to specify include paths and the include paths themselves here. The complete contents of this entry will replace the entry `%I` in the Options (see above).

- *Obj. extension*

  Enter the object extension to be used for the compiler output.

- *List of files*

  – *Default*

    Restores the default values.

  – *Add*

    A file selection dialog is opened where you can select the file which should be additionally compiled.

  – *Remove*

    Removes the item you have selected in the list box.

  – *Modify*

    A dialog is displayed and the selected entry can be modified

  – *Clear*

    Deletes all the items in the list box.

- *Dependencies*

  List the dependencies for the makefile here. Note that the name of the additional file itself is automatically generated as a dependency.

Linker

To link all the compiled files (generated ones, the ones building the library and additional ones) you must configure a linker.

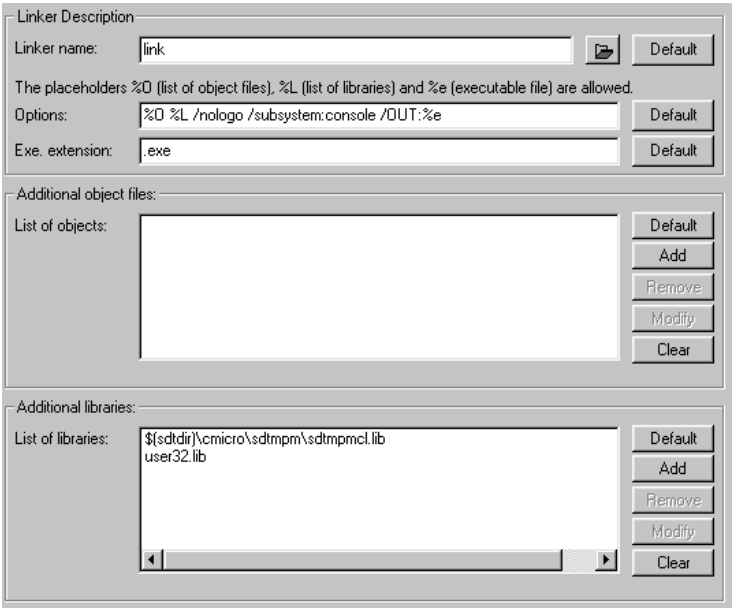


Figure 538: Linker configuration

Click *Default* if you want to restore the default values.

- *Linker name*

  Enter the name of the linker application.

- *Options*

  Enter the options given to the compiler as command line arguments here.

  – Enter the dummy parameter `%O` in the place of the list of all compiled files and the additional object files to link (used for the makefile generation).
  – Enter the dummy parameter `%L` in the place of the list of all the libraries (if there are some) which have to be linked.
  – Enter the dummy parameter `%e` in the place where the executable file's name has to be inserted (used for the makefile generation).

- *Exe. extension*

  Enter the executable extension to be used for the linker output.

- *Additional object files* and *Additional libraries*

  – *Default*

  Restores the default values.

  – *Add*

  A file selection dialog is opened where you can select the file which should be additionally linked to the executable.

  – *Remove*

  Removes the item you have selected in the list box.

  – *Modify*

  A dialog is displayed and the selected item in the list box can be modified.

  – *Clear*

  Deletes all the items in the list box.

Library Manager

> **Note:**
> This section is available only if an SDL and/or ASN.1 coder is selected. Please see "Communication" on page 2952 for details on how to select a coder.

The library manager offers the command that will be used to build a library from all the coder files. The dialog is shown in Figure 539

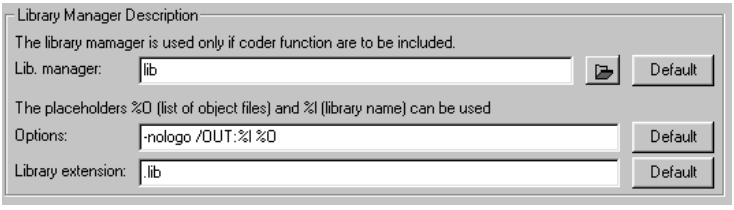


Figure 539: Library Manager Configuration

Note:

If no library manager is given all the coder object files will be linked directly to the target application.

Click *Default* if you want to restore the default values.

- *Lib. Manager*

  Enter the name of the library manager application.

- *Options*

  Enter the options given to the library manager as command line arguments here.

  – Enter the dummy parameter `%O` in the place of the list of all coder object files to link (used for the makefile generation).
  – Enter the dummy parameter `%l` in the place where the library's name has to be inserted (used for the makefile generation).

- *Library extension*

  Enter the library extension here.

Make

You can select the make tool to do all the compile and link actions here.

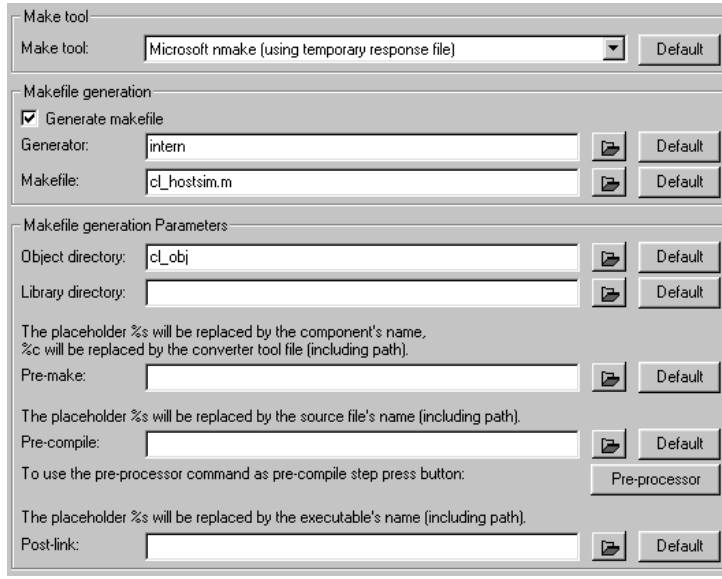


Figure 540: Make tool configuration

Click *Default* if you want to restore the default values.

- *Make tool*

  Select the make tool you wish to use. Currently the following make tools are supported: (see "Make Applications" on page 3003)

  – Microsoft nmake (using temporary response file)
  – Microsoft nmake (ignore exit codes)
  – Microsoft nmake
  – UNIX make
  – Tasking mk166
  – VxWorks gnu tool chain make

Hint:

"Using temporary response file" means that the compiler's and the linker's command line options are passed to the compiler/linker via a temporary file generated by the make tool.

This feature can only be used if the compiler in use supports this facility.

Note:

If the make tool you would like to use is not in the list. Please add a new definition in `sdttaex.par`. For more information view "Make Applications" on page 3003

- *Generate makefile*

  Select this if a makefile should be generated before make is invoked. The name of the generated makefile is the one in *Makefile*.

- *Generator*

  The Targeting Expert is designed to execute external makefile generators whenever the build-in makefile generator (`intern`) is not sufficient.

  Enter the name (and the path) of the external makefile generator here. Please see "External Makefile Generator" on page 3009 for further information about how to build an external makefile generator.

- *Makefile*

  Enter the name of the makefile.

- *Object directory*

  The object directory is the relative path seen from the current target directory, in which the compiler output files (object files) should be written.

Note:

If the directory `<target_directory>/<object_directory>` does not exist it will be automatically created.

- *Library directory*

The library directory is the absolute path to the target library to be used. Under normal circumstances this field can be left empty as the Targeting Expert automatically uses the path in the installation:

`<installationdir>/sdt/sdtdir/<platform_sdtdir>`

Note:

The Cmicro target library will be expected in a sub-directory called `cmicro`.

The Cadvanced library will be expected in a sub-directory called `INCLUDE`.

The Cextreme library will be expected in a sub-directory called `cextreme`.

- *Pre-make*

  The pre-make step will be executed before any other action specified in the makefile is performed.

  – The placeholder `%s` will automatically be replaced by the component's name.

- *Pre-compile*

  The pre-compile of the generated file(s) will be executed before they are compiled.

  Enter the application which performs the pre-compile here.

  – The placeholder `%s` can be used to represent the source files name.

  – The preprocessor button inserts the complete command that is necessary to pre-process the generated C file with the default compiler before the compilation with the target compiler.

Hint:

It is probably useful to use the utility functions delivered in combination with the Targeting Expert. Please see "Utilities" on page 3010

- *Post-link*

The post-link of the linked files will be executed as the last step in the make task.

Enter the application which performs the post-link here. (E.g. a conversion from absolute to Intel-HEX format).

– The placeholder `%s` can be used to represent the linker output file.

Configure and Scale the Target Library

The configuration and the scaling of the used target library is done by setting/resetting compiler macros (`#define` in C). These macros will be called “flags” in the further description.

For more information concerning allowed flags see

- `sct_mcf.h` (Cadvanced), see “Some Configuration Macros” on page 3156.
- `ml_mcf.h` (Cmicro), see “Compilation Flags” on page 3486 in chapter 66, *The Cmicro Library*
- `extreme_user_cfg.h` (Cextreme), see Optimization and Configuration.

The Targeting Expert offers an easy way of setting/resetting these flags. All the allowed flags are divided into different groups, e.g. SDL support and environment.

Figure 541 shows an example for Cmicro.

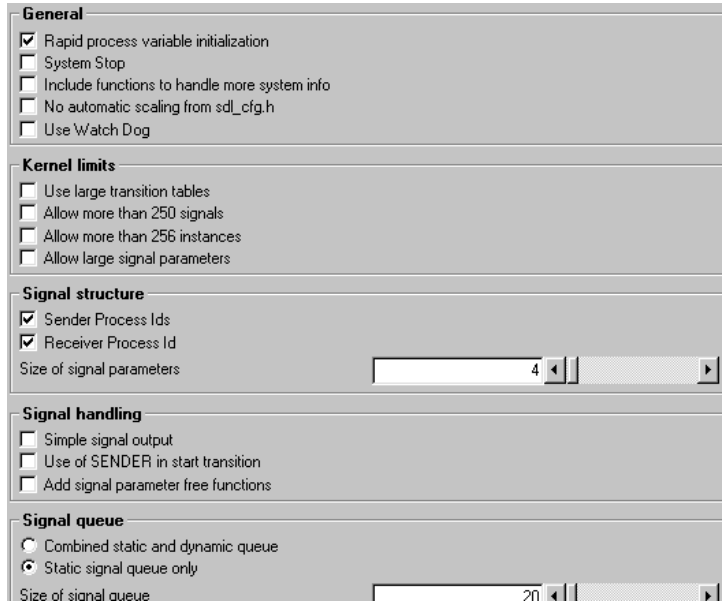


Figure 541: Configuration flag dialog

When all the configurations have been done the Targeting Expert generates

- `ml_mcf.h` (a C header file for Cmicro) which will be included in `ml_typ.h` during the compilation of the corresponding files.
  This is done for all the pre-defined integrations using the SDL to C compiler Cmicro.
- `sct_mcf.h` (a C header file for Cadvanced) which will be included in `scttypes.h` if the flag `USER_CONFIG` is defined.
  The flag `USER_CONFIG` is only defined for the pre-defined integrations Application (applclenv) and Application, debug (debclenv-com) using the SDL to C compiler Cadvanced.
- `extreme_user_cfg.h` (a C header file for Cextreme) which will be included in `extreme_kern.h` during compilation.

Configure the SDL Target Tester (Cmicro only)

The configuration of the SDL Target Tester is very similar to the configuration of the target library.

The settings done for the target will also be generated into the file `ml_mcf.h`. (Please see "Configure and Scale the Target Library" on page 2946.)

Configure the Host (Cmicro only)

The settings for the host will be generated into the file `sdtmt.opt` which will be read by the SDL Target Tester during start-up to get information about the gateway to be used and the target's memory layout. Please see "Communication Setup on the Host System" on page 3615 in chapter 67, *The SDL Target Tester* for more information about how to configure the SDL Target Tester's host application.

4. Actions to perform:
   - Configure how to Make the Component
   - Make the Component

Configure how to Make the Component

The possible options are divided into four pages:

> **Note:**
>
> All the Analyzer options entered in the Organizer will be re-used by the Targeting Expert.
>
> When a warning or error is found during analysis a warning message will popup. By setting the Organizer preference Organizer*ShowLogLevel to Never this dialog will be suppressed.

SDL to C Compiler

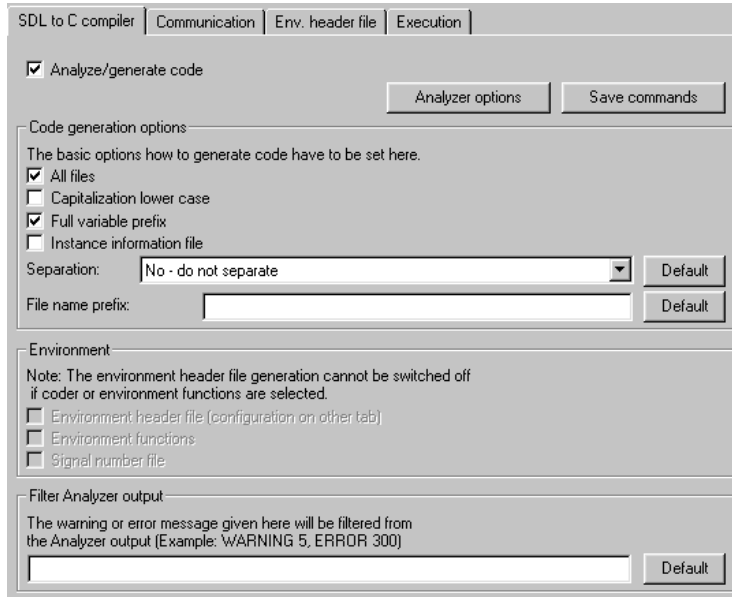


Figure 542

Note:

Because the Advanced Mode is switched off there might be some controls disabled because it does not make sense to modify them for the selected integration.

- *Analyze/generate code*

  Switch the Analyzer and SDL to C compiler execution on/off when starting the make process.

- *Analyzer options*

  The Organizer's Analyzer-Options dialog is displayed and the Analyzer options used for all the integrations can be modified. Please see "Analyze SDL" on page 113 in chapter 2, *The Organizer*.

- *Save commands*

A text file containing all the Analyzer commands configured for the selected integration can be saved. It is possible to start the Analyzer directly using this command file with `sdtsan < commandfile`

- *All files*

  If selected, the SDL to C compiler will generate all files anew. It does not matter whether the file's contents have changed or not. Unselecting it can economize the build process.

- *Capitalization lower case*

  When selected all identifiers are translated to lower case. Otherwise capitalization is used in the declaration of the object

- *Full variable prefix*

  All the variables used in SDL will be generated with a full prefix to C to make them unique if this item is selected. This comes with a disadvantage of long variable names.

- *Instance information file*

  An instance information file will be generated. Please see "SDL Instance Information" on page 2512 in chapter 54, *The SDL Analyzer*.

- *Separation*

  It is possible to select between three alternatives:

  – do not separate
  – use the user defined separation done in Organizer
  – separate each SDL entity into one C file

- *File name prefix*

  All the generated files (except those with fixed names like `sdl_cfg.h`) will be generated with the given file name prefix.

- *Environment header file*

  The environment header file (`.ifc`) which is used in the coder and environment functions will be generated if switched on.

- *Environment functions*

  Environment functions will be generated or not.

Note:

If the generation of environment functions is switched on it may be needed to add the environment source file to Source Files.

Please see

- "Initializing the Environment / Interface to the Environment" on page 3532 in chapter 66, *The Cmicro Library* for Cmicro
- "Building an Application" on page 2767 for Cadvanced

- *Signal number file* (Cadvanced only)

  Generate a signal number file if selected.

- *Conversion style* (Cmicro only)

  Select the rules how to generate the converter file here.

- *Filename* (Cmicro only)

  Give a name to the converter file

- *Amount of comments* (Cextreme only)

  Selects the amount of comments that the code generator will generate.

- *Generate run time tests* (Cextreme only)

  Controls if run time tests should be generated.

- *Generate sdt references* (Cextreme only)

  Adds comments with sdt references used to navigate from code to model.

- *Generate trace code* (Cextreme only)

  Controls if trace code should be generated.

Communication

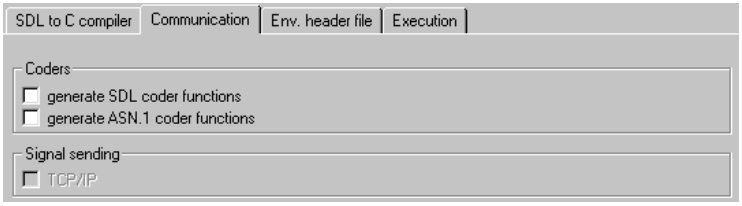


Figure 543: Communication configuration

- *Generate/do not generate SDL coder functions.*
  (Please see "Type description nodes for SDL types" on page 2791 in chapter 57, *Building an Application*.)
- *Generate/do not generate ASN.1 coder functions.*
  (Please see "ASN.1 Type Information Generated by ASN.1 Utilities" on page 2841 in chapter 58, *ASN.1 Encoding and De-coding in the SDL Suite*.)
- *TCP/IP signal sending*

Note:

The TCP/IP communication is supported only for a threaded integration using the SDL to C compiler Cadvanced.

The signal sending to other components can be done via a communications link. If it is switched on, a wizard dialog to set up all the needed information pops up (see Figure 544).

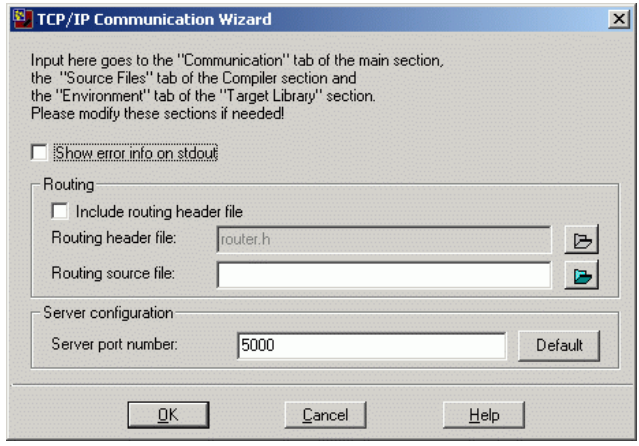


Figure 544: TCP/IP communication wizard

As described in the dialog there are settings done in different sections of the Targeting Expert:

- the manual configuration file `sct_mcf.h`
- the list of source files to compile, i.e. the makefile

Furthermore the generation of

- an Environment header file
- Environment functions and
- SDL coder functions

will be switched on.

Caution!

If the TCP/IP communication is switched **on,** a routing source file has to be provided, so that the signals destination can be calculated correctly! Otherwise all the signals will be sent to the sending application.

Caution!

If the TCP/IP communication is switched **off** the routing source file needs to be removed from the list of source files to compile by hand!

Env. Header File

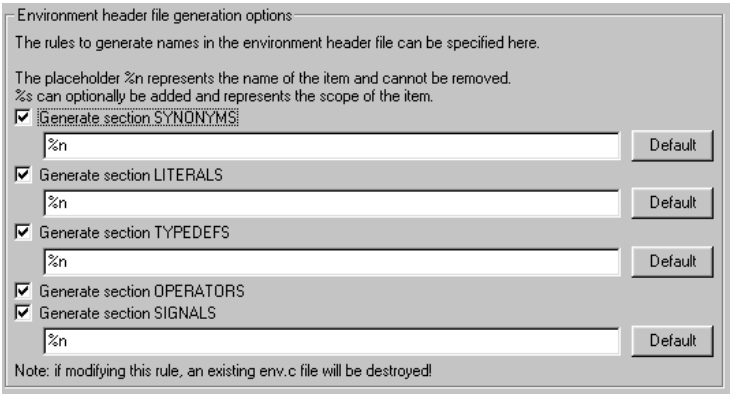


Figure 545: Env. header file configuration

The way the environment header file(`.ifc` file) is generated can be configured here. Please see "System Interface Header File" on page 2777 in chapter 57, *Building an Application* for more information.

Note:

The CHANNELS section cannot be configured for Cmicro because the SDL systems structure (blocks and channels) will be lost after code generation.

Execution

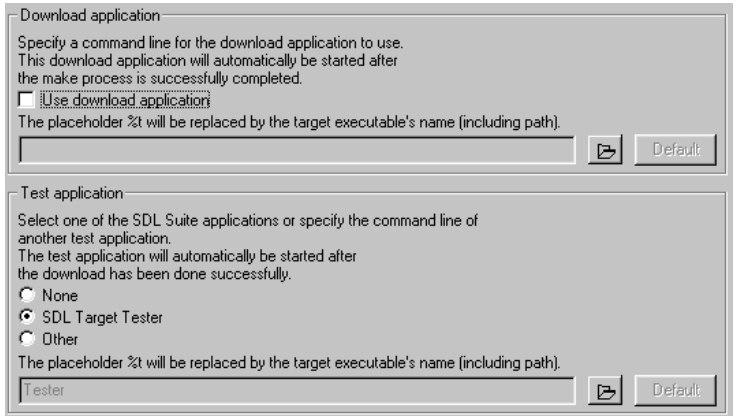


Figure 546: Download and test tool configuration

- *Download Application*

  Enter also the complete command line which will be necessary to invoke the download application. You can use the placeholder `%t` which will automatically be replaced by the name of the target executable's name (including the full path).

Note:

The selected download application will automatically be started if make was successful. The Targeting Expert does not try to invoke a download application if "Use download application" is un-checked.

- *Test Application*

  There are two kinds of test applications:

  – The applications that are part of the SDL Suite, i.e. *"The SDL Simulator" on page 2129*, *"The SDL Explorer" on page 2297* and "The SDL Target Tester" on page 3601.
  – The applications from other vendors, e.g. a debugger or emulator.

  If you select *Other* you have to enter also the complete command line for the desired test tool. You can use the placeholder `%t` which

will automatically be replaced by the target executable (including the full path).

Make the Component

The whole make process consists of several different tasks which will automatically be executed one after another. These tasks are:

- Analyze and Generate Code
- Generate a Makefile
- Compile and link
- Execute (target application or the application in Download Application and/or Test Application)

The execution of the next task will be stopped if the current task returns with an error. The Targeting Expert event log will give information about the process' state.

Analyze and Generate Code

The selected SDL to C compiler will be invoked and code will be generated for the selected component.

There are two possibilities:

- make
  In that case only code for the modified parts of the SDL system will be generated.
- full make
  Code for the complete SDL system is generated. Even if there are no modifications.

Generate a Makefile

The makefile to be used will be generated. Please see "Make" on page 2942 for information on how to give the makefile's name and how to switch the makefile generation on or off.

> **Hint:**
>
> If the build-in makefile generation does not fit your needs it is possible to customize the makefile generation.
>
> Please see "External Makefile Generator" on page 3009 to get information on how to build a makefile generator.

Compile and link

The external make tool will be invoked using the generated makefile. Please see "Make" on page 2942 for information on how to select the make tool.

> **Hint:**
>
> To force a compilation of all the C source files the object files have to be deleted first. Please use the *Clean* entry in the Make Menu.

Execute

The target executable will be executed.

> **Note:**
>
> It is sometimes not possible to invoke the target executable from the Targeting Expert, e.g. if the target executable is built for a micro controller. In this case you can select a download application or any other test tool (e.g. debugger) which will be executed instead. Please see "Download Application" on page 2955 and "Test Application" on page 2955.

> **Hint:**
>
> For running test cases the target executable's output to `stdout` and `stderr` will be re-directed automatically into the Targeting Expert event Log. The target simply should be executed by entering "`%t`" as the test application (Please see "Test Application" on page 2955).

Batch Mode

Using the Targeting Expert in the batch mode means starting `taexbatch` instead of `sdttaex`.

The commands have to be given one after another by hand or in a command file. (Please see an Example of a Batch File.) In this case the Targeting Expert has to be started with `taexbatch < commandfile`

> **Note:**
>
> If the commands are given in a command file the Targeting Expert should be started with one of the options `-yes` or `-no` to answer all upcoming questions automatically. Otherwise the application will probably hang up.

Syntax of Batch Mode Commands

Command Names

You may abbreviate a command name by giving sufficient characters to distinguish it from other command names. A hyphen ('-') is used to separate command names into distinct parts.

Abbreviation of Commands

You may abbreviate any part as long as the command does not become ambiguous.

> **Hint:**
>
> Commands used in command files should not be abbreviated because future implementations may conflict with those abbreviations.

Parameters in Commands

Parameters are separated by one or several spaces. Parameters containing spaces have to be given in quotation marks.

Case Insensitivity in Commands

In command names there is no distinction made between upper and lower case letters.

Some more Detailed Description of Parameter Types

- `<flagname>`

  Allowed flag names depend on the SDL to C compiler in use.

  – For Cadvanced a description of allowed flags can be found in section "Some Configuration Macros" on page 3156.
  – For Cmicro see "Compilation Flags" on page 3486 in chapter 66, *The Cmicro Library* for a more detailed description.
  – For Cextreme see Optimization and Configuration.

  `<flagname>` parameters are case sensitive.

- `<boolvalue>`

  Allowed boolean values can be `true`, `on` and `yes` which all have the same meaning no matter if written lower case or not. The same applies for the values `false`, `off` and `no`.

- `<stringvalue>`

  Any string value is allowed here. There is no check at all if the value can be interpreted correctly. E.g. if "abc" is entered although and integer value is needed it will not be checked but probably cause a compilation error afterwards.

- `<entry>`

  Allowed entries depend on the class used in the command. Please see `<class>`. Parameters of type `<entry>` are case insensitive.

- `<class>`

  Allowed classes and the depending entries are shown in the table below. Parameters of type `<class>` are case insensitive.

  For all allowed combinations of classes and entries the commands Get-Setting and Default-Setting can be used.

  – Compiler

allowed entries	function to set contents
AsCpp	Set-Setting-String
CodInclude	Set-Setting-String
FilesToCompile	Add-Setting-String
Include	Set-Setting-String
LibFlag	Set-Setting-String
Options	Set-Setting-String
Options2	Set-Setting-String
Options3	Set-Setting-String
TesterFilesToCompile	Set-Setting-String
Tool	Set-Setting-String
Tool2	Set-Setting-String
Tool3	Set-Setting-String

– Linker

allowed entries	function to set contents
LibrariesToLink	Add-Setting-String
ObjectsToLink	Add-Setting-String
Options	Set-Setting-String
Tool	Set-Setting-String

– AddCompiler

allowed entries	function to set contents
AddFilesToCompile	Add-Setting-String
Depend	Set-Setting-String
Include	Set-Setting-String
Options	Set-Setting-String
Tool	Set-Setting-String

– Make

allowed entries	function to set contents
Tool	Set-Setting-String

– Global

allowed entries	function to set contents
AddObjectExtension	Set-Setting-String
CodeGenerator	Set-Setting-String
ExecutableExtension	Set-Setting-String
FileNamePrefix	Set-Setting-String
FullSeparation	Set-Setting-Bool
GenerateASNCoder	Set-Setting-Bool
GenerateEnvFunctions	Set-Setting-Bool
GenerateEnvHeader	Set-Setting-Bool
GenerateIfcChannels	Set-Setting-Bool
GenerateIfcLiterals	Set-Setting-Bool
GenerateIfcOperators	Set-Setting-Bool
GenerateIfcSignals	Set-Setting-Bool
GenerateIfcSynonyms	Set-Setting-Bool
GenerateIfcTypedefs	Set-Setting-Bool

allowed entries	function to set contents
GenerateInstanceInfo	Set-Setting-Bool
GenerateLowerCase	Set-Setting-Bool
GenerateMakefile	Set-Setting-Bool
GenerateSDLCoder	Set-Setting-Bool
GenerateSignalNumber	Set-Setting-Bool
IfcPrefixChannels	Set-Setting-String
IfcPrefixLiterals	Set-Setting-String
IfcPrefixSignals	Set-Setting-String
IfcPrefixSynonyms	Set-Setting-String
IfcPrefixTypes	Set-Setting-String
LibraryDirectory	Set-Setting-String
MakefileGenerator	Set-Setting-String
MakefileName	Set-Setting-String
PreCompile	Set-Setting-String
PreMake	Set-Setting-String
PostLink	Set-Setting-String
ObjectDirectory	Set-Setting-String
ObjectExtension	Set-Setting-String
TestTool	Set-Setting-String

Description of Batch Mode Commands

In this section, the batch mode commands are listed in alphabetical order, along with their parameters and a description.

Add-PR

Parameters:

<pr-filename>

Further PR files can be specified here which should be considered by the SDL to C compiler.

The first PR file has to be given by using Open-PR command.

Add-Setting-String

Parameters:

`<class> <entry> <stringvalue>`

The list `<entry>` will be extended for the item `<stringvalue>`.

Add-UserSection-String

Parameters:

`<stringvalue>`

The `<stringvalue>` will be added in the user section of `ml_mcf.h` (Cmicro), `sct_mcf.h` (Cadvanced) or `extreme_user_cfg.h` (Cextreme).

Analyze

Parameters:

`<none>`

The SDL system will be analyzed.

Append-Setting-String

Parameters:

`<class> <entry> <stringvalue>`

`<stringvalue>` will be appended to the current value of `<class> <entry>`.

Clear-UserSection

Parameters:

`<none>`

The complete user section of `ml_mcf.h` (Cmicro), `sct_mcf.h` (Cadvanced) or `extreme_user_cfg.h` (Cextreme) will be removed.

Default-Setting

Parameters:

`<class> <entry>`

The default value of the given `<entry>` will be restored.

Exit
Parameters:
<none>

The Targeting Expert exits (same as Quit).

Generate-Code
Parameters:
<none>

A code generation for the selected component is done using the SDL to C compiler selected in the associated settings.

Generate-Code-Full
Parameters:
<none>

A full code generation for the selected component is done using the SDL to C compiler selected in the associated settings.

Generate-Makefile
Parameters:
<none>

A makefile will be generated for the selected component. The make-file's name is taken from the associated settings.

Get-Setting
Parameters:
<class> <entry>

The current contents of <entry> will be printed on the screen.

Include
Parameters:
<batch-filename>

All the commands listed in <batch-filename> will be processed.

Help
Parameters:
<none>

A list of all commands with their needed parameters is printed.

Make-All
<none>

For the application or all the nodes and components that are configured (combined or separate configuration) the steps of analysis, code generation, makefile generation and compilation/linkage will be performed.

Make-Clean

Parameters:

`<none>`

All the object files will be removed.

Make-Selected

Parameters:

`<none>`

The make tool will be executed using the associated make tool and the associated makefile of the selected component.

Open-PDM

Parameters:

`<filename>`

The partitioning diagram file `<filename>` will be loaded.

Open-PR

Parameters:

`<filename>`

The PR file `<filename>` will be loaded and internally converted into a partitioning diagram model.

Open-SDT

Parameters:

`<filename>`

The system file `<filename>` will be loaded and internally converted into a partitioning diagram model.

Prepend-Setting-String

Parameters:

`<class> <entry> <stringvalue>`

`<stringvalue>` will be prepended to the current value of `<class> <entry>`.

Qualifier

Parameters:

`<SDL-qualifier>`

The given <SDL-qualifier> will be used for building a part of the SDL system.

Quit

Parameters:

<none>

The Targeting Expert exits (same as Exit).

Replace-Setting-String

Parameters:

<class> <entry> <old-stringvalue> <new-stringvalue>

The string <old-stringvalue> of <entry> will be replaced with <new-stringvalue>.

Save-Settings

Parameters:

<none>

The modified settings of all the available applications, nodes and components will be saved.

Select

Parameters:

[<application-node-component>]

The entry specified by <application-node-component> in the partitioning diagram will be selected. This step has to be executed before you can generate.

One of the commands Open-PDM, Open-PR or Open-SDT has to be executed before.

Set-Compiler

Parameters:

<compilername>

The given <compilername> will be set as the used compiler. The <compilername> must be available in the default settings of the Set-Integration.

The command Set-Integration has to be executed before.

Set-Flag-Bool

Parameters:

<flagname> <boolvalue>

The SDL to C compiler's library flag <flagname> will be set/reset due to the <boolvalue>.

> **Caution!**
>
> There is no check for interdependencies to other flags.

Set-Flag-String

Parameters:

```
<flagname> <stringvalue>
```

The SDL to C compiler's library flag <flagname> will be set to the <value>.

> **Caution!**
>
> There is no check for upper and lower limits. You must be aware to set a value that is in the allowed range!

Set-Integration

Parameters:

```
<integration>
```

The pre-defined integration settings <integration> will be set.

- Simulations
  - Performance Simulation
  - Realtime Simulation
  - Simulation
  - TTCN Link
- Validations
  - Validation
- TargetTest
  - Application DEBUG CA
  - Application DEBUG CE
  - HostSimulation
  - HostSimulation (CL)
  - HostSimulation (CLENV)
- Light Integrations
  - Application CA
  - Application CE
  - Application CM
  - Application TEST

- Threaded Integrations
  - OSE
  - Solaris
  - VxWorks
  - POSIX/Linux
  - Win32

Note:

To set a Realtime Simulation for example, the `<integration>` has to be given in quotation marks, i.e.

```
Set-Integration "Realtime Simulation"
```

Caution!

In batch mode there is no check if the selected integration fits to the component settings!

Please have a look to "Distributed Pre-defined Integration Settings" on page 2979 for more information.

The command Select has to be executed before.

Set-Setting-Bool

Parameters:

`<class> <entry> <boolvalue>`

The boolean `<entry>` will be set to its new value.

Set-Setting-String

Parameters:

`<class> <entry> <stringvalue>`

The non-boolean `<entry>` will get the new content `<stringvalue>`.

Start-Download

Parameters:

`<none>`

The download application selected in Download Application will be invoked.

Start-Testtool

Parameters:

`<none>`

The test application specified in Test Application will be invoked.

system

Parameters:

`<systemcommand>`

Executes the `<systemcommand>` on the underlying OS. The following placeholders can be used in `<systemcommand>` and will be replaced before execution:

placeholder	... will be replaced by...
`%s`	system directory
`%b`	target directory (the one given in the Organizer)
`%t`	sub target directory (the one calculated depending on the selected component and integration)
`%e`	executable name (inclusive extension)
`%i`	intermediate directory

Target

Parameters:

`<target>`

The `<target>` becomes the SDL system's part to build. The generated files will be placed in a subfolder with `<target>` as part of the folder name.

Write-Log

Parameters:

`<filename>`

All the output to `stdout` and `stderr` will also be written into the file `<filename>`.

Example of a Batch File

The following batch file can be used to build a Simulator using the Targeting Expert batch mode.

Example 489: Simple batch file ––––––––––––––––––––––––––––––

```
Open-SDT access.sdt
select access-node-component
```

```
set-integration Simulation
set-Compiler Microsoft
Generate-Code
Generate-Makefile
Make-selected
Exit
```

Note:

The items `application`, `node` and `component` depend on the partitioning diagram model used.

Internal

Partitioning Diagram Model File

The Targeting Expert needs to know how the SDL system which is used for targeting should be partitioned. This is done by using a partitioning diagram model.

> **Note:**
>
> The partitioning diagram model will be generated by the Deployment Editor for deployed systems into a partitioning diagram model file `<partitioning diagram model>.pdm`. It should not be modified by hand! Please see "Generating Partitioning Diagram Data for the Targeting Expert" on page 1787 in chapter 40, *The Deployment Editor* for more information.
>
> If there is no deployment done for a system, the Targeting Expert generates a default partitioning diagram model for its internal use. This partitioning diagram model is not available as a file. For a default partitioning diagram model there is no chance to specify integration models or threads.

Definitions

The complete SDL system is mirrored in the partitioning diagram's *application* which takes several *nodes*.

Each *node* can represent a run-time (physical) object with memory and processing capability and is a collection of several *components*.

Each *component* is a collection of different *objects*, with each *object* representing an SDL qualifier. A *component* will result in one executable or OS task.

File Syntax Example

Estimated there is an SDL system like the one shown in Figure 547,

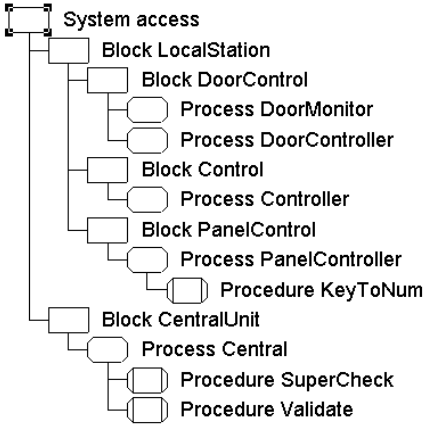


Figure 547: An SDL system

then the simplest possible partitioning diagram model looks like the one shown in Figure 548.

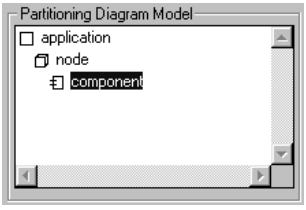


Figure 548: A possible partitioning diagram model

The first few lines of the corresponding partitioning diagram model file is shown in Example 490:

Example 490: Partitioning diagram model file ––––––––––––––––––––

```
TIMESTAMP: 948787119

APPLICATION*0: access
detail+SystemFileName: access.sdt

NODE*1: node

COMPONENT*2: component
```

```
OBJECT*3: object
detail+Qualifier: access
-------------------------------------------------------
```

Explanations

- The `TIMESTAMP` entry is needed for future implementations.
- The `APPLICATION*0: access` gives the name of the used SDL system. The `0` is used as an identifier for internal purposes.
- `detail+SystemFileName` is the name of the used `<system-name>.sdt` file
- `detail+PRFileName` is the name of the used `<systemname>.pr` file. Subsequent entries of `detail+PRFileName` can be used to insert more than one PR file.
- `NODE*1: node` specifies that the following `components` belong to the node. The node is called `node` and the unique id `1` is used for internal purposes.
- `COMPONENT*2: component` specifies that the following `objects` belong to this component. The component is called `component` and the unique id `2` is used for internal purposes.
- `OBJECT+5: object` gives one further object which will be assigned to the current `component`.
- `detail+Qualifier: access` means that all the SDL entities that are sub entities of the qualifier access (in this case the complete system) will be part of the `object`.

More Supported Keywords

Details of COMPONENT

- `detail+Integration: Light|Threaded` specifies in which integration model the component should be build. For backwards compatibility, the keyword `ThreadedLight` is accepted as an alias for `Threaded`.

Details of OBJECT

- `detail+Type: system|block|process` tells which kind of SDL agent the qualifier is all about.

Details of THREAD

- `THREAD: name` is a sub entry of `COMPONENT` and specifies a thread, e.g. for a threaded integration.
- `detail+ThreadPriority: value` specifies the OS priority of the particular thread.
- `detail+StackSize: value` specifies the stack size of the thread. (bytes)
- `detail+QueueSize: value` specifies the maximum number of signals that are allowed in the queue.
- `detail+MaxSignalSize: value` specifies the maximum size of an individual signal in the queue. (bytes)
- `detail+OneThreadPerInstance: true|false` tells if each process instance will become its own thread in a threaded integration if the value is `true`.

Configuration Files

For each SDL to C compiler that will be used there is one configuration file `ca_conf.def` (Cadvanced), `cm_conf.def` (Cmicro) and `ce_conf.def` (Cextreme).

The configuration files for all SDL to C compilers can be found in `<installationdir>/sdt/sdtdir`.

The `c*_conf.def` files are ASCII text files that can be (should be) extended by you to adapt a new compiler or to adapt a new communications link (Cmicro only).

You can find all the known configuration flags in:

- "Some Configuration Macros" on page 3156 (Cadvanced)
- "Compilation Flags" on page 3486 in chapter 66, *The Cmicro Library* (for Cmicro)

Structure of the Configuration File cm_conf.def

Note:

All the descriptions in this section are valid for the SDL to C compiler Cmicro only.

Compiler

All the compilers currently supported by a SDL to C compiler library are listed in the c*_conf.def file like displayed in Example 491.

Example 491: Structure of a compiler entry

```
BEGIN
  IAR_C51
  IAR systems compiler of 8051
  COMPILER
  Microcontroller compiler
END
```

The first and the last line of the example simply marks the start and the end of one entry.

Line 2 is the flag which will be used to include the appropriate compiler section in the used library (see `ml_typ.h` for Cmicro and `scttypes.h` for Cadvanced).

Line 3 is the text which will be shown in the Targeting Experts user interface.

Line 4 must be `COMPILER` in this case.

Line 5 is the compiler group this compiler should be associated with. In the distributed files `c*_conf.def` there are three groups, namely:

- `Micro controller compiler`
- `PC compiler`
- `Workstation compiler`

Communications Link

Note:

This section is valid only if the Cmicro SDL to C Compiler is used.

The example below shows the structure of one entry.

Example 492: Structure of a `cm_conf.def` entries

```
 1 BEGIN
 2   XMK_USE_V24
 3    V24 communication
 4    COMMLINK
 5    V24 interface
 6   AUTOSET
 7     XMK_V24_BAUD
 8     XMK_V24_DEVICE
 9   END
10 END
```

Lines 1 and 10 mark the start and the end of one entry.

Line 2 gives the name of the flag as it appears in the generated file `ml_mcf.h`. According to this example line 3 is the marker which appears in the Targeting Expert graphical user interface.

Line 4 specifies the main group this entry belongs to. It has to be `COMMLINK` in this case.

Line 5 gives the subgroup of the entry. This subgroup can be anything you like. In the delivered version there is only the subgroup `V24 interface`. However, you can define your own communications link `CAN Bus` for example.

The lines 6 and 9 belong together and between these lines all flags are listed which have to be set when the flag on line 2 is set. The amount of flags (lines 7 and 8 in the example) is free. In the same way as `AUTOSET` in line 6 there can also be the entries `RESET` or `DEPEND`.

`RESET` means the following flags will be reset if this flag is defined and `DEPEND` means this flag can only be set if the following flags (lines 7 and 8) are already defined.

Not given in the example above is the section `VALUE`, (like `AUTOSET` finished by an `END`) which assign a value to the flag. This section can include three lines of information. The first gives the default value. And, if the value is numeric, the second line gives the lowest and the third one the highest value. You are asked to have a look into `c*_conf.def`.

Also not given in the example above is the section `VALUE_LIST` (finished by an `END`) which assign a value to the flag with a list of allowed values given. The first value is the default value, the following values

(with no upper limit) are giving the list of allowed values. The default value must be given in the list again.

Pre-defined Integration Settings

Introduction

Pre-defined integration settings must be seen as a set of default values which can be used for targeting purposes.

There are several pre-defined integration settings distributed with each installation.

- Each set of predefined integration settings is stored in an ASCII file with the extension `.its`.
- The amount of pre-defined integration settings that are to be handled by the Targeting Expert can simply be enlarged by copying a new file `<name>.its` or by exporting own settings. (See "Export" on page 2922)
- The integration names have to be unique!
- The Targeting Expert searches for `.its` files in the following directories:

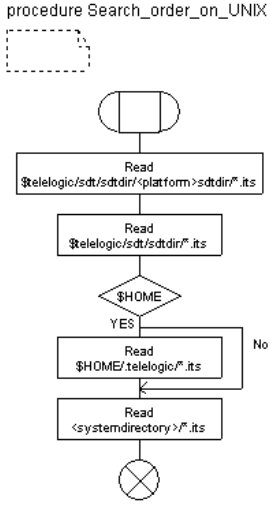


Figure 549: Search order on UNIX

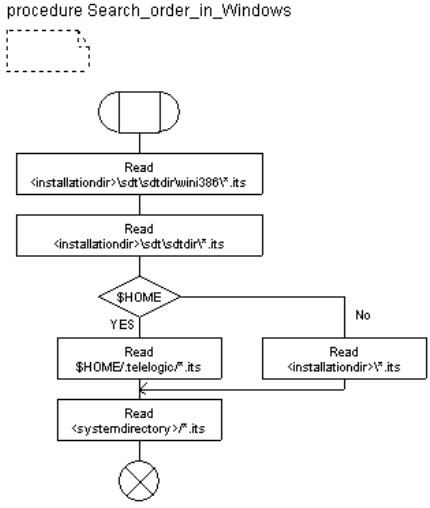


Figure 550: Search order in Windows

Distributed Pre-defined Integration Settings

For Use with Cadvanced SDL to C Compiler

- Simulation (`debcom`)

Supported compiler	On Sun: – Sun GNU gcc – Sun Workshop cc – Sun Workshop CC In Windows: – Microsoft VC++
Used kernel directory	-
Timers	Timers are not implemented, i.e. timers set in the SDL system will expire at once
Environment	Will be handled in the Simulator UI
More info	See chapter 49, *The SDL Simulator*.

- Realtime Simulation (`debclcom`)

Supported compiler	On Sun: – Sun GNU gcc – Sun Workshop cc – Sun Workshop CC On Windows: – Microsoft VC++
Used kernel directory	-
Timers	Timers are implemented in a way that one timer tick is equal to one second, i.e. a timer set to 10 in the SDL system will expire after 10 seconds.
Environment	Will be handled in the Simulator UI
More info	See chapter 49, *The SDL Simulator*.

- Performance Simulation (`perfsim`)

Supported compiler	On Sun: – Sun GNU gcc – Sun Workshop cc – Sun Workshop CC In Windows: – Microsoft VC++
Used kernel directory	-
Timers	Not handled
Environment	Not handled
More info	See chapter 63, *The Performance Library*.

- TTCN Link

Supported compiler	On Sun: – Sun GNU gcc – Sun Workshop cc On Windows: – Microsoft VC++
Used kernel directory	`$(sdtdir)/SCT*TTCNLINK`
More info	See chapter 35, *TTCN Test Suite Generation*.

- Validation

Supported compiler	On Sun: – Sun GNU gcc – Sun Workshop cc On Windows: – Microsoft VC++
Used kernel directory	`$(sdtdir)/SCT*VALIDATOR`
More info	See chapter 52, *The SDL Explorer*.

- Application (`applclenv`)

Supported compiler	On Sun: – Sun GNU gcc – Sun Workshop cc – Sun Workshop CC On Windows: – Microsoft VC++
Used kernel directory	-
Timers	Timers are implemented in a way that one timer tick is equal to one second, i.e. a timer set to 10 in the SDL system will expire after 10 seconds.
Environment	Has to be implemented by the user.
More info	See "Compilation Switches" on page 3119 in chapter 61, *The Master Library*.

- Application, debug (`debclenvcom`)

Supported compiler	On Sun: – Sun GNU gcc – Sun Workshop cc – Sun Workshop CC On Windows: – Microsoft VC++
Used kernel directory	-
Timers	Timers are implemented in a way that one timer tick is equal to one second, i.e. a timer set to 10 in the SDL system will expire after 10 seconds.
Environment	Has to be implemented by the user.
More info	See "Compilation Switches" on page 3119 in chapter 61, *The Master Library*.

- Threaded integrations for the platforms
  - OSE (Softkernel Solaris and Windows)
  - Solaris
  - VxWorks (Softkernel Solaris and Windows)

Note: (VxWorks only)

The environment variable `WIND_BASE` has to be set on your system!

- Win32
- OSE (Softkernel Solaris)

Note: (OSE only)

The environment variable `OSE_ROOT` has to be set on your system!

Supported compiler	OSE: – Sun GNU gcc – Microsoft VC++ Solaris: – Sun GNU gcc – Sun Workshop cc – Sun Workshop CC VxWorks: – gnu tool chain Win32: – Microsoft VC++
Used kernel directory	-
Timers	OS timer
More info	See "Threaded Integration" on page 3303 in chapter 64, *Integration with Operating Systems*.

For Use with the Cmicro SDL to C Compiler

- Application (`applclenv`)

Supported compiler	On Sun: – Sun GNU gcc – Sun Workshop cc – Sun Workshop CC On Windows: – Microsoft VC++
Used kernel directory	-

Timers	Timers are implemented in a way that one timer tick is equal to one second, i.e. a timer set to 10 in the SDL system will expire after 10 seconds.
Environment	Has to be implemented by the user.
Template files	Taken from $(sdtdir)\cmicro\template
More info	The target executable can be executed on a command line.

- Application including Target Tester and communication via sockets

Supported compiler	On Sun: – Sun GNU gcc – Sun Workshop cc – Sun Workshop CC On Windows: – Microsoft VC++
Used kernel directory	-
Timers	Timers are implemented in a way that one timer tick is equal to one second, i.e. a timer set to 10 in the SDL system will expire after 10 seconds.
Environment	Has to be implemented by the user.
Template files	Taken from $(sdtdir)\cmicro\template
More info	The target executable has to be started manually after the Target Tester is started by the Targeting Expert.

- Host Simulation (`debcom`)

Supported compiler	On Sun: – Sun GNU gcc – Sun Workshop CC On Windows: – Microsoft VC++
Used kernel directory	-

Timers	Timers are not implemented, i.e a timer set in the SDL system will never expire.
Environment	Will be handled in the SDL Target Tester UI.
Template files	Taken from $(sdtdir)\cmicro\template
More info	The target is executed as the SDL Target Tester's gateway. After a complete make The SDL Target Testerwill be started and the target can be tested.

- Real-time Host Simulation (`debclcom`)

Supported compiler	On Sun: – Sun GNU gcc – Sun Workshop CC On Windows: – Microsoft VC++
Used kernel directory	-
Timers	Timers are implemented in a way that one timer tick is equal to one second, i.e. a timer set to 10 in the SDL system will expire after 10 seconds.
Environment	Will be handled in the SDL Target Tester UI.
Template files	Taken from $(sdtdir)\cmicro\template
More help	The target is executed as the SDL Target Tester's gateway. After a complete make The SDL Target Tester will be started and the target can be tested.

- Real-time Host Simulation with environment (`debclenvcom`)

Supported compiler	On Sun: – Sun GNU gcc – Sun Workshop CC On Windows: – Microsoft VC++
Used kernel directory	-

Timers	Timers are implemented in a way that one timer tick is equal to one second, i.e. a timer set to 10 in the SDL system will expire after 10 seconds.
Environment	Has to be implemented by the user.
Template files	Taken from $(sdtdir)\cmicro\template
More help	The target is executed as the SDL Target Tester's gateway. After a complete make The SDL Target Tester will be started and the target can be tested.

For Use with the Cextreme SDL to C Compiler

- Application (applclenv)

Supported compiler	On Sun: – Sun GNU gcc – Sun Workshop cc – Sun Workshop CC On Windows: – Microsoft VC++ On Linux: – Linux GNU gcc
Timers	Timers are implemented in a way that one timer tick is equal to one second, i.e. a timer set to 10 in the SDL system will expire after 10 seconds.
Environment	Has to be implemented by the user
More info	See Optimization and Configuration

- Application, debug (debclenvcom)

Supported compiler	On Sun: – Sun GNU gcc – Sun Workshop cc – Sun Workshop CC On Windows: – Microsoft VC++ On Linux: – Linux GNU gcc
Timers	Timers are implemented in a way that one timer tick is equal to one second, i.e. a timer set to 10 in the SDL system will expire after 10 seconds.
Environment	Has to be implemented by the user
More info	Debug options for code generation and compiler settings enabled by default. See Optimization and Configuration

- Threaded integrations for the platforms
  - Posix
  - Win32
  - VxWorks (Windows)

Supported compiler	On Sun: – Sun GNU gcc – Sun Workshop cc – Sun Workshop CC On Windows: – Microsoft VC++ On Linux: – Linux GNU gcc On VxWorks: – GNU tool chain
Timers	Use OS timers
Environment	Has to be implemented by the user

More info	WIND_BASE has to be set for VxWorks. Other options, see Optimization and Configuration. A deployment diagram is needed to specify what the code generator should generate threads for, see The Deployment Editor.

Syntax of the .its Files

There is a more or less complex syntax to be followed in `.its` files which will be described here. In principle each `.its` file is divided into four sections:

- The Description about the Settings
- The Settings which are once more split into
  - the Global Settings (A)
  - the Compiler specific Definitions
- The Configuration Settings to Be Set
- The Configuration Settings to Be Reset

You can find further syntax information in “Comments and empty Lines” on page 2995 and “Other Rules” on page 2996.

Description about the Settings

The Targeting Expert will read in the description and display it in the user interface whenever a set of pre-defined integration settings is selected.

This section starts with `[DESCRIPTION]` and has got the only entry `HelpLink`. The description is a link name to this manual.

The Settings

The section of the `.its` files which contains the settings starts with `[SETTINGS]`. They must be ordered with the Global Settings (A) first, followed by the Compiler specific Definitions.

- Global Settings (A)

The global settings start with the `<keyword1>` equal to `Preferences` and the delimiter ’+’, i.e. an allowed line in this section looks like:

```
Preferences+<keyword2>: parameter
```

Example 493: Pre-defined integration settings (2A)

```
Preference+IntegrationSettings: <name>
```

The meaning of `<keyword2>` can be found in the table below.

Keyword 2	Meaning of the parameter
`IntegrationSettings`	The name of this pre-defined integration setting is entered here.

- Global Settings (B)

The global settings part B start with the `<keyword1>` equal to the name of the pre-defined integration settings (see Global Settings (A)) and the delimiter '+', i.e. an allowed line in this section looks like:

```
<name>+<keyword2>: parameter
```

Example 494: Pre-defined integration settings (2B)

```
Simulation+CompilersUsed:       Microsoft
Simulation+CodeGenerator:       Cadvanced
Simulation+GenerateMakefile:    YES
```

The meaning of `<keyword2>` can be found in the table below.

Keyword 2	Meaning of the parameter
`CompilersUsed`	The name of a compiler which is used to build the target application must be entered here. This line can appear more than once per file to allow the description of more than one compiler.
`CodeGenerator`	The name of the SDL to C compiler this pre-defined integration settings are designed for must be entered to the right of this keyword. Currently the SDL to C compilers – Cadvanced – Cmicro – Cextreme are possible here.

Keyword 2	Meaning of the parameter
GenerateAllFiles	All the files will be re-generated even if the contents have not changed (YES) or not (NO)
GenerateMakefile	The makefile generation should be switched on (YES) but can of course be switched off by specifying NO here.
GenerateLowerCase	The SDL to C compiler should generate all SDL identifiers in lower case (YES) or as defined (NO)
GenerateEnvFunctions	The SDL to C compiler should generate template environment functions (YES) or not (NO)
GenerateEnvHeader	The SDL to C compiler should generate an environment header file (YES) or not (NO)
GeneratePrefix	All variables will be generated with full prefix. Default is (YES) but can of course be switched off by specifying NO here.
GenerateSignalNumber	The SDL to C compiler should generate a signal number file (YES) or not (NO)
GenerateSDLCoder	The SDL to C compiler should generate an SDL coder functions file (NO, ASCII or <user>)
GenerateASNCoder	The SDL to C compiler should generate an ASN.1 coder functions file (NO, BER, PER or <user>)
Separation	The SDL to C compiler shall generate the C files separated according to the given value: NO, USER, FULL
Analyze	The Analyzer/SDL to C compiler should be invoked during the make (YES) or not (NO)
TestTool	The Test application to be used can be entered here, i.e. the entries Simulator, Validator or Tester but also a complete command line.
FileNamePrefix	The SDL to C compiler will generate all files with the given filename prefix.

Keyword 2	Meaning of the parameter
GenerateIfcSynonyms	The SDL to C compiler should generate the SYNONYMS section into the env. header file (YES) or not (NO). (depends on GenerateEnvHeader)
GenerateIfcLiterals	The SDL to C compiler should generate the LITERALS section into the env. header file (YES) or not (NO). (depends on GenerateEnvHeader)
GenerateIfcTypedefs	The SDL to C compiler should generate the TYPEDEFS section into the env. header file (YES) or not (NO). (depends on GenerateEnvHeader)
GenerateIfcOperators	The SDL to C compiler should generate the OPERATORS section into the env. header file (YES) or not (NO). (depends on GenerateEnvHeader)
GenerateIfcSignals	The SDL to C compiler should generate the SIGNALS section into the env. header file (YES) or not (NO). (depends on GenerateEnvHeader)
GenerateIfcChannels	The SDL to C compiler should generate the CHANNELS section into the env. header file (YES) or not (NO). (depends on GenerateEnvHeader)
IfcPrefixSynonyms	The SDL to C compiler will generate SYNONYMS using the specified rule (Default is %n)
IfcPrefixLiterals	The SDL to C compiler will generate SYNONYMS using the specified rule (Default is %n)
IfcPrefixTypes	The SDL to C compiler will generate SYNONYMS using the specified rule (Default is %n)
IfcPrefixSignals	The SDL to C compiler will generate SYNONYMS using the specified rule (Default is %n)
IfcPrefixChannels	The SDL to C compiler will generate SYNONYMS using the specified rule (Default is %n)

Keyword 2	Meaning of the parameter
SuppressText	The Analyzer error or warning message to suppress can be given here.

- Compiler specific Definitions

This segment has got all the definitions for an unlimited amount of compilers. All the keywords start with

```
<name of the setting>+<compiler-name>*
```

followed by one or two further keywords. If it is followed by two keywords both are separated by '~'.

Example 495: Pre-defined integration settings (3) ----------

```
Simulation+Microsoft*Compiler~Tool:    cl
Simulation+Microsoft*Compiler~Options: -nologo %I -ML -c -D_Windows -DSCTDEBCOM
/Fo%o %s
Simulation+Microsoft*Compiler~Include: -I$(sctuseinclude)
Simulation+Microsoft*Compiler~Flag:   IC86
Simulation+Microsoft*Linker~Tool:     link
Simulation+Microsoft*Linker~Options:  -nologo -subsystem:console %O /OUT:%e
Simulation+Microsoft*ObjectExtension:      _smc.obj
Simulation+Microsoft*ExecutableExtension: _smc.exe
Simulation+Microsoft*FileToCompile:   $(sdtdir)\INCLUDE\sctsdl.c
Simulation+Microsoft*FileToCompile:   $(sdtdir)\INCLUDE\sctpred.c
Simulation+Microsoft*FileToCompile:   $(sdtdir)\INCLUDE\sctos.c
Simulation+Microsoft*FileToCompile:   $(sdtdir)\INCLUDE\sctmon.c
Simulation+Microsoft*FileToCompile:   $(sdtdir)\INCLUDE\sctutil.c
Simulation+Microsoft*FileToCompile:   $(sdtdir)\SCTADEBCOM\sctpost.c
Simulation+Microsoft*LibrariesToLink: $(sdtdir)\INCLUDE\msvc50\libpost.lib
Simulation+Microsoft*LibrariesToLink: user32.lib
Simulation+Microsoft*Make~Tool: Microsoft nmake (using temporary response file)
Simulation+Microsoft*MakefileName:        my_makefile.m
Simulation+Microsoft*MakefileGenerator:   -
Simulation+Microsoft*ObjectDirectory:     objects
```

All the keywords and the allowed combinations can be found in the table below.

Keyword 3	Keyword 4	Meaning of the parameter
Compiler	Tool	Name of the compiler application
Compiler	Tool2	Name of the C parser application (optional)
Compiler	Tool3	Name of the assembler application (optional)

Keyword 3	Keyword 4	Meaning of the parameter
Compiler	Options	Compiler command line options (Please see "Compiler" on page 2975)
Compiler	Options2	C parser command line options (optional)
Compiler	Options3	Assembler command line options (optional)
Compiler	LibFlag	The compiler's option to set defines plus the library flag (e.g. SCTDEBCOM)
Compiler	Include	The compiler's option for include paths and the include paths
Compiler	CodInclude	The compiler's option for include paths and the include paths to the coder files
Compiler	Flag	Compiler flag (Please see "Compiler Flag" on page 2936)
Linker	Tool	Name of the linker application
Linker	Options	Linker command line options (Please see "Linker" on page 2939)
ObjectExtension		Extension of the object files
ExecutableExtension		Extension of the target executable
FilesToCompile		Kernel, library and template files to be compiled. This entry can be used more than once.
TesterFilesToCompile		Tester files to be compiled. This entry can be used more than once. (Cmicro only)
FilesToCopy		Template files which should be copied into the target directory. Environment variable $(sdtdir) can be used
ObjectsToLink		Additional object files to be linked. This entry can be used more than once.

Keyword 3	Keyword 4	Meaning of the parameter
LibrariesToLink		Additional libraries to be linked (e.g. the compiler's libraries). This entry can be used more than once.
Make	Tool	The name of the make application. This can be: • Microsoft nmake (using temporary response file) • Microsoft nmake • UNIX make
MakefileName		The name of the makefile to be generated (or respectively to be used) is entered here.
MakefileGenerator		The Targeting Experts supports the use of external makefile generators which is to be entered here. An intern means that the build-in makefile generator is to be used
ObjectDirectory		The relative path (seen from the target directory) for the object directory can be entered here. If not specified Targeting expert will generate the makefile in a way that the object files will be written into the target directory when make is executed.
LibraryDirectory		If this entry is specified the SDL to C compiler's library will be taken from the specified path. If not specified it will be taken from $sdtdir (%SDTDIR%).
Download	Tool	The full command line to invoke the download application.
PreMake		Action to perform before the target application will be made in the makefile.

Keyword 3	Keyword 4	Meaning of the parameter
PreCompile		If the generated code should be modified before it is compiled, an appropriate application (complete command line) should be entered here.
PostLink		If the linked target application needs to be modified (e.g. conversion into HEX format), the appropriate application (complete command line) should be entered here.
AddCompiler	Tool	For the additional files (see FilesToCompile) the name of the compiler application must be entered here.
AddCompiler	Options	The command line options for the compiler in AddCompiler must be specified here.
AddCompiler	Include	The compiler's option for include paths and the include paths for the compiler entered in AddCompiler
AddCompiler	Depend	The dependencies for the additional files (used inside of the generated makefile).
AddObjectExtension		The object extension of the AddCompiler must be entered here.
AddFilesToCompile		Kernel files to be compiled (e.g. communications link). This entry can be used more than once.

Default Configuration Settings

For several pre-defined integrations it is necessary to have some configuration flags defined as default. In this section a simple list of all configuration flags which should be set is given.

This section starts with [CONFIGURATION_DEFAULT].

Please see *"Configuration Files" on page 2974* for more information about configuration flags.

Configuration Settings to Be Set

For several pre-defined integrations it is necessary to have some configuration flags set without giving you the chance to un-select them. In this section a simple list of all configuration flags which have to be set is given.

This section starts with `[CONFIGURATION_SET]`.

Please see *"Configuration Files" on page 2974* for more information about configuration flags.

Configuration Settings to Be Reset

In opposite to the configuration flags which have to be set, it is of course necessary sometimes to prevent the selection of other flags. This can be done in this section by simply listing all the flags that have to be reset.

This section starts with `[CONFIGURATION_RESET]`.

Please see *"Configuration Files" on page 2974* for more information about configuration flags.

SDL to C Compiler Settings to Be Disabled

For some pre-defined integrations it does not make sense to configure all the SDL to C compiler options available. It is possible here to list all the options to be disabled in the user interface.

This section starts with `[SDL_TO_C_COMPILER_DISABLED]`.

SDL to C Compiler Settings to Be Enabled

For some pre-defined integrations some SDL to C compiler options needs to be enabled by default. It is possible here to list all the options to be enabled in the user interface.

This section starts with `[SDL_TO_C_COMPILER_DEFAULT]`.

Comments and empty Lines

- Empty lines are allowed everywhere in the an `.its` file.
- Comments are allowed everywhere in the `.its` file but have to start with an '#' in the first column.

Other Rules

- Only one set is allowed per `.its` file
- The amount of compilers supported for each set is unlimited.
- The characters '[', ']', '+', '*', '~' and ':' are forbidden in compiler names or the name of the pre-defined integration settings itself.

User-defined Integration Settings

All the settings done by the user will be stored in separate files into the target directory, i.e.:

- The complete set of settings will be stored (not the difference to the defaults)
- There is one file for each integration configuration done for an application, node or component
- If several nodes or components have to use the same settings it is possible to manually select one file for them.
  Please see "Handling of Settings" on page 2920.

Target Sub-Directory Structure

The Targeting Expert automatically sets up a sub-directory structure for all the applications, nodes and components. Furthermore there is a sub directory for each kind of integration done.

Example 496: Sub-directory structure ----------

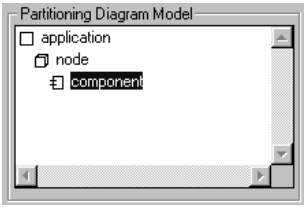


Figure 551: Example of a partitioning diagram

Estimated there is a partitioning diagram which looks like Figure 551, the Targeting Expert generates a sub-directory structure like:

```
<target_dir>
   + application._0
       + node._1
           + component._2
```

Note:

The suffixes `._*` give unique identifiers which make it possible to rename the application, nodes or components and to keep the done settings. In this case the Targeting Expert automatically modifies the names of the subdirectories.

Later, after you have built a Simulator and an Explorer for the `component`, for example, then the sub-directory structure was extended to:

```
<target_dir>
     + application._0
          + node._1
              + component._2
                   + Simulator
                   + Validator
```

Flat Directory Structure

When using an `.sdt` or `.pr` file as input the directory structure will be flatten automatically. I.e it will look like

```
<target_dir>
     + application._0
          + Simulator
          + Validator
```

This is possible because there can only be one node with one component in both cases.

Generated Makefile

Before the selected make tool is invoked the Targeting Expert generates a makefile with all of the necessary settings done. (Please see "Configure Compiler, Linker and Make" on page 2930 for information on how to set up the makefile generation.)

Caution!

The standard header files of the used SDL to C compiler library are **not** used as dependencies in the compilation rules, i.e. after modifying one of these files you have to select the *Clean* entry in the Make Menu.

Comparisons

There are a few differences that need to be mentioned when comparing the way the Targeting Expert handles makefiles with the way the Organizer's Make dialog does.

When using the Targeting Expert...

- ... there is no makefile (`systemname.m`) generated by the SDL to C compiler. Instead it generates a sub-makefile (`component_gen.m`) which is automatically parsed and inserted into the Targeting Expert makefile, i.e. the Targeting Expert still gets all the information from the SDL to C compiler but in a different way.
- ... there is not support of template makefiles by the Targeting Expert. The makefiles generated by the Targeting Expert can be modified directly in the so called "user sections". Please see "User Modifications" on page 2999.
- ... there are no `makeoptions/make.opt` files (taken from the kernel directories) used at all. Instead of this, the Targeting Expert gets all the make information from the Pre-defined Integration Settings and places them into its makefile.
- ... the `comp.opt` files placed in the kernel directories are not used at all! Again all the information is taken from the Pre-defined Integration Settings.
- ... the `sccd` cannot be used for pre-processing purposes as default because the Targeting Expert does not use `sccd.cfg` files from the kernel directories. It is recommended to use the Targeting Expert Preprocessor instead.

Note:

The only exceptions in which the Targeting Expert uses some information from the associated kernel directory are the integrations Validation and TTCN Link. In both cases a library is taken from the kernel directory.

User Modifications

After all it is possible that there are a few adaptations to be done, e.g. if there is an assembler file which needs to be assembled and linked.

Example 497 shows the section of a generated makefile where it is possible to add further object files.

Example 497: Generated makefile (1)

```
######## UserObjectsStart (Do not edit this line!) ##########
userOBJECTFILES1 =

userOBJECTFILES2 =

######## UserObjectsEnd   (Do not edit this line!) ##########
```

The list `userOBJECTFILES1` will be inserted at the front of all the object files to link. Accordingly `userOBJECTFILES2` will be inserted at the end.

The rules and the dependencies for the files in Example 497 must be given in the section shown in Example 498.

Example 498: Generated makefile (2)

```
######## UserRulesStart (Do not edit this line!) ##########

######## UserRulesEnd   (Do not edit this line!) ##########
```

Caution!

Any modification that is done outside of the sections described in Example 497 and Example 498 will be overwritten when the makefile is generated anew.

Parameter File sdttaex.par

General

One of the delivered configuration files used by the Targeting Expert is called `sdttaex.par`. Several different sections give information that is statically used during runtime. Under normal circumstances this file does not need to be modified. However, there might be the need to extend the Targeting Expert functions.

Please see the following sub-sections for more information what can be configured:

- Compiler Error Descriptions
- Preprocessor Commands
- Make Applications
- C++ Options
- Debug Options
- Compiler Dependent Defaults
- Restricted Compilers
- Editor Commands
- Additional Files
- Host Configuration Options (used for Cmicro only)

Search Order for sdttaex.par

The following shown search order applies when the Targeting Expert is started. The distributed version of `sdttaex.par` is always placed in `<installationdir>/bin/<platform>bin`

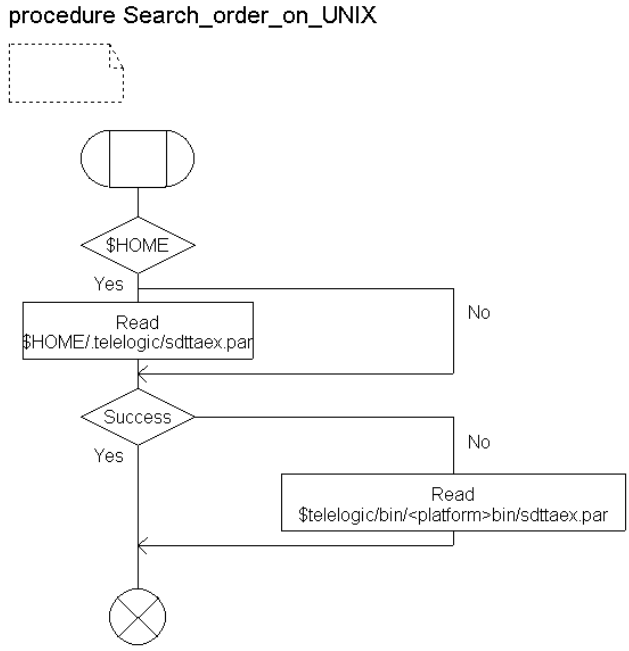


Figure 552: Search order for sdttaex.par on UNIX

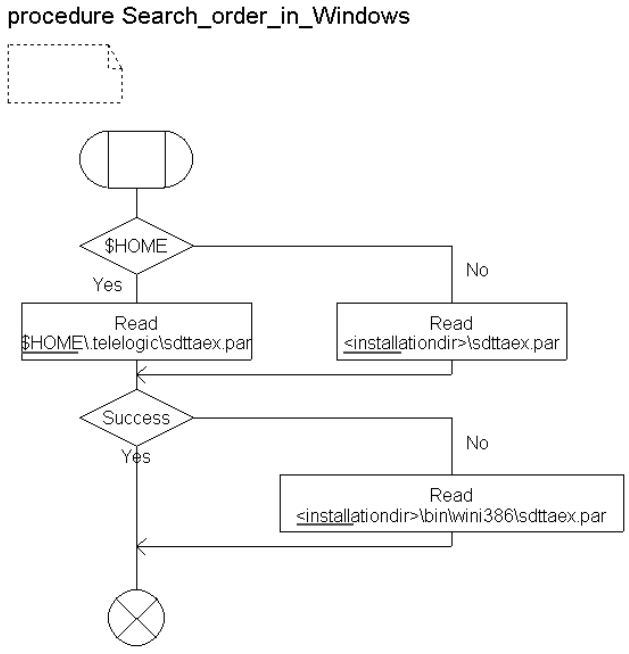


Figure 553: Search order for sdttaex.par in Windows

Compiler Error Descriptions

The section `[COMPILER-ERROR-DESCRIPTION]` of `sdttaex.par` is filled with regular expressions describing the construction of compiler error messages.

Example 499: `[COMPILER-ERROR-DESCRIPTION]`

```
[COMPILER-ERROR-DESCRIPTION]
{cl}       {1} {.*([1234567890]}       {0} {2} {([1234567890]*)}       {1} {1}
{cc166}    {2} {.*[A-Za-z]:}           {0} {1} {[1234567890]*: }       {0} {2}
{gcc}      {1} {.*:[1234567890]}       {0} {2} {:[1234567890]*:}       {1} {1}
{icc8051}  {1} {.*,[1234567890]}       {1} {3} {,[1234567890]* }       {1} {1}
{c51}      {1} {OF .*[CH]: }           {3} {2} {LINE [1234567890]* OF} {5} {3}
{cc}       {1} {.*, l}                 {1} {4} {line [1234567890]*:}   {5} {1}
{CC}       {1} {.*, l}                 {1} {2} {line [1234567890]*:}   {5} {1}
{aCC}      {1} {: .*,}                 {3} {2} {line [1234567890]* #}  {5} {2}
{ccpentium} {1} {.*:[1234567890]}      {0} {2} {:[1234567890]*:}       {1} {1}
{icpp}     {1} {!E .*(}                {3} {1} {(.*)}                  {1} {1}
```

Each line gives the description for one compiler. The eight entries have the meaning shown below (explained on the gcc entry):

1. The name of the compiler executable
2. The amount of lines the compiler error message takes
3. The regular expression to determine the erroneous file
4. The amount of characters thrown away at the beginning of the found file name
5. The amount of characters thrown away at the end of the found file name. (In this example the colon and the numeric character do not belong to the file name but both are needed to find it.)
6. The regular expression to determine the line number
7. The amount of characters thrown away at the beginning of the found line. (In this example the colon at the front of the regular expression does not belong to the line number but is needed to find it).
8. The amount of characters thrown away at the end of the found line. (In this example the colon at the end of the regular expression does not belong to the line number but is needed to find it).

Preprocessor Commands

This section takes the preprocessor commands for all the default compilers that can be set in the preferences. It has got the following contents:

Example 500: `[PREPROCESSOR-COMMANDS]`

```
[PREPROCESSOR-COMMANDS]
{Sun GNU gcc}     {gcc -P -E -C}
{Sun Workshop cc} {cc -C -P}
{Sun Workshop CC} {CC -E}
{Microsoft}       {cl -P -EP -C -nologo}
```

Make Applications

The section `[MAKE_APPLICATIONS]` of `sdttaex.par` is used to specify the command line of the make application, i.e. how it will be invoked.

Example 501: `[MAKE_APPLICATIONS]`

```
[MAKE_APPLICATIONS]
{Microsoft nmake}                                {nmake /f}    {}        {}
{Microsoft nmake (ignore exit codes)}            {nmake /i /f} {}        {}
{Microsoft nmake (using temporary response file)} {nmake /f}   {@<<\n\t} {\n<<}
```

```
{Tasking mk166}                                  {mk166 -f}    {}        {}
{VxWorks make}                                   {make -f}     {}        {}
{UNIX make}                                      {make -f}     {}        {}
```

Each line is the description for one make application. The four entries (given between '{' and '}') have the following meaning.

1. The text to identify it in the make configuration. See "Make tool" on page 2943
2. The name of the make application and the option used to specify the makefile.
3. This is the start sequence for the generation of temporary response files. Empty for most make applications.
4. This is the stop sequence for the generation of temporary response files. Empty for most make applications.

C++ Options

The C++ options specify what has to be added to the compiler's Options and the linker's Options to compile and link the SDL to C compiler's library as C++ code.

Example 502: `[C++-OPTIONS]`

```
[C++OPTIONS]
{Sun GNU gcc} {-xc++}  {-lstdc++}
{Microsoft}   {-TP}    {}
```

Each line consist of the three entries

1. compiler name
2. compiler command line option
3. linker command line option

Debug Options

The debug options specify what has to be added to the compiler's Options and the linker's Options to compile and link the target application including debug information.

Example 503: `[DEBUG_OPTIONS]`

```
[DEBUG-OPTIONS]
{Sun GNU gcc}     {-g}              {-g}
{Sun Workshop cc} {-g}              {-g}
{Sun Workshop CC} {-g}              {-g}
```

```
{Microsoft}        {-D "_DEBUG"}     {/debug}
{icc12}            {-g}              {-g}
```

Each line consist of the three entries

1. compiler name
2. compiler command line option
3. linker command line option

Compiler Dependent Defaults

The compiler dependent defaults ease the implementation of new pre-defined integrations because the values given here do not need to appear again in the .its file(s).

Example 504: `[COMPILER-DEPENDENT-DEFAULTS]`

```
[COMPILER-DEPENDENT-DEFAULTS]
{Borland}   {bcc32} {bcc32} {tlib} {.obj} {.exe} {.lib} {%I -c -w- -DUSING_DLL -
DIC86 -o%o %s} {-e%e %L %O} {%l /C /E /P256 %O} {-I$(sctCODERDIR)} {bcc_obj}
{Borland make (using temporary response file)}
```

The different entries mean the following (listed from left to right):

1. Compiler name
2. Compiler tool name
3. Linker tool name
4. Library manager name
5. Object file extension
6. Executable file extension
7. Library file extension
8. Compiler options
9. Linker options
10. Library Manager Options
11. Coder include path
12. Relative path to intermediate directory
13. make application to use per default

Restricted Compilers

There are a few target compilers which do not support file names which are longer that 8.3 (8 character in front of '.' and a 3 character extension). To ensure that files are generated only which fit to the 8.3 requirement. This section looks as follows:

Example 505: `[8.3-COMPILERS]` ----------

```
[8.3-COMPILERS]
{c51}
{icpp}
{icc12}
```

Editor Commands

The Targeting Expert allows to use other text editor different to the build-in one. The information how to start these editors is given in the section `[EDITOR-COMMANDS]` shown below.

Example 506: `[EDITOR-COMMANDS]` ----------

```
[EDITOR-COMMANDS]
{Windows} {TextPad}            {textpad -ca -g "%f"(%l,0)}
{Windows} {UltraEdit}          {UEdit32 %f/%l/0}
{Windows} {Codewright}         {codewright %f -G%l}
{Windows} {MS DeveloperStudio} {msdev %f}
{UNIX}    {Emacs}              {emacs %f}
{UNIX}    {dtpad}              {/usr/dt/bin/dtpad %f}
{UNIX}    {nedit}              {nedit -line %l %f}
```

There are 2 placeholders used in the command. `%f` will be replaced by the name of the file to open and `%l` gives the line number which should be shown.

Additional Files

The sections shown in Example 507 will used to find out which files belong to external parts of the distributed SDL to C compiler libraries.

Example 507: Additional Files Sections ----------

```
[CMICRO-KERNEL-FILES]
$(sctkerneldir)/mk_main.c
$(sctkerneldir)/mk_sche.c
$(sctkerneldir)/mk_queu.c
$(sctkerneldir)/mk_outp.c
$(sctkerneldir)/mk_tim1.c
$(sctkerneldir)/sctpred.c

[CMICRO-TI-KERNEL-FILES]
$(sctkerneldir)/ti_sche.c
$(sctkerneldir)/ti_queu.c
$(sctkerneldir)/ti_outp.c
$(sctkerneldir)/ti_tim1.c
$(sctkerneldir)/ti_init.c
$(sctkerneldir)/sctpred.c
```

```
[TESTER-FILES]
$(scttesterdir)/mt_tsdl.c
$(scttesterdir)/mt_tsys.c
$(scttesterdir)/mt_cod.c
$(scttesterdir)/mt_cmd.c
$(scttesterdir)/mt_opt.c
$(scttesterdir)/mt_deb.c
$(scttesterdir)/mt_rec.c
$(scttesterdir)/mt_play.c
$(sctkerneldir)/ml_buf.c

[TI-TESTER-FILES]
$(scttesterdir)/mt_tsdl.c
$(scttesterdir)/mt_tsys.c
$(scttesterdir)/mt_cod.c
$(scttesterdir)/mt_cmd.c
$(scttesterdir)/mt_opt.c
$(scttesterdir)/mt_deb.c
$(sctkerneldir)/ml_buf.c

[CODER-FILES]
$(sctCODERDIR)/cucf_er.c
$(sctCODERDIR)/cucf_er_sdt.c
$(sctCODERDIR)/bms/bms.c
$(sctCODERDIR)/bms/bms_small.c
$(sctCODERDIR)/ems/ems.c
$(sctCODERDIR)/ems/ems_eo_sdt.c
$(sctCODERDIR)/er/ascii/ascii.c
$(sctCODERDIR)/er/ber/ber_base.c
$(sctCODERDIR)/er/ber/ber_content.c
$(sctCODERDIR)/er/ber/ber_decode.c
$(sctCODERDIR)/er/ber/ber_encode.c
$(sctCODERDIR)/er/per/per_base.c
$(sctCODERDIR)/er/per/per_content.c
$(sctCODERDIR)/er/per/per_decode.c
$(sctCODERDIR)/er/per/per_encode.c
$(sctCODERDIR)/er/mms/mms.c
$(sctCODERDIR)/vms/vms_base.c
$(sctCODERDIR)/vms/vms_check.c
$(sctCODERDIR)/vms/vms_export.c
$(sctCODERDIR)/vms/vms_print.c

[TCP-IP-FILES]
$(sctTCPIPDIR)/tcpipcomm.c
```

Note:

The [TESTER-FILES] and [TI-TESTER-FILES] only apply for the Cmicro SDL to C compiler.

The [TCP-IP-FILES] only applies for the Cadvanced SDL to C compiler.

If the environment variable SDTTAEXIGNOREDEFAULTFILES is set, the contents of the sections [CMICRO-KERNEL-FILES], [CMICRO-TI-KERNEL-FILES], [TESTER-FILES] and [TI-TESTER-FILES] are ignored by the Targeting Expert (the files listed are not included in the resulting Makefiles). This enables the customer to manage the complete list of files in custom (.its) files.

Host Configuration Options

The host configuration options are the default host configurations for different compilers when using the Target Tester, i.e. these options are only used for Cmicro applications. An example can be found below.

Example 508: [HOST-CONFIGURATION-OPTIONS] **(Borland compiler)**

```
[HOST-CONFIGURATION-OPTIONS]
bcc32 "UNIT-NAME            sec"
bcc32 "UNIT-SCALE           1.0"
bcc32 "LENGTH_CHAR          1"
bcc32 "LENGTH_SHORT         2"
bcc32 "LENGTH_INT           4"
bcc32 "LENGTH_LONG          4"
bcc32 "LENGTH_FLOAT         4"
bcc32 "LENGTH_DOUBLE        8"
bcc32 "LENGTH_POINTER       4"
bcc32 "ALIGN_CHAR           8"
bcc32 "ALIGN_SHORT          8"
bcc32 "ALIGN_INT            8"
bcc32 "ALIGN_LONG           8"
bcc32 "ALIGN_FLOAT          8"
bcc32 "ALIGN_DOUBLE         8"
bcc32 "ALIGN_POINTER        8"
bcc32 "ENDIAN_CHAR          1"
bcc32 "ENDIAN_SHORT         21"
bcc32 "ENDIAN_INT           41"
bcc32 "ENDIAN_LONG          41"
bcc32 "ENDIAN_FLOAT         41"
bcc32 "ENDIAN_DOUBLE        81"
bcc32 "ENDIAN_POINTER       41"
```

External Makefile Generator

General

As it is probably necessary for you to modify the layout of the generated makefile, the source code of the external makefile generator “makegen” is delivered with every distribution. It can be found in `<installationdir>/sdt/sdtdir/util/<platform>`.

Source and Make Files

The source files delivered are:

- `makegen.[ch]`
- `ini_api.h`
- `pdm_api.h`

Furthermore there are makefiles and libraries:

- cc - compiler **on UNIX**
  - `makegencc.m`
  - `makegenlib.a`
- gcc 2.95.2 - compiler **on UNIX**
  - `makegengcc.m`
  - `makegenlib.a`
- Microsoft VC++ 6.0 compiler **in Windows**
  - `makegencl.m`
  - `makegencl.lib`

Utilities

General

Delivered in combination with the Targeting Expert there is an application called `taexutil`. This application can be used directly from the Targeting Expert *Tools* menu or on the command line, e.g. in makefiles.

The following utility functions are supported:

- DOS to UNIX
- UNIX to DOS
- Indent
- Preprocessor

Note:

Each utility will create a backup file called `<inputfile>.bak`

DOS to UNIX

This utility can be used to modify ASCII files. It replaces all the found '`\r\n`' sequences against '`\n`'.

- Command line:
  `taexutil D2U <inputfile> <outputfile>`
  `<inputfile>` and `<outputfile>` can be equal but both must be given.

UNIX to DOS

This utility can be used to modify ASCII files. It replaces all the found '`\n`' characters against '`\r\n`' sequences.

- Command line:
  `taexutil U2D <inputfile> <outputfile>`
  `<inputfile>` and `<outputfile>` can be equal but both must be given.

Indent

The indentation of the given ASCII file will be corrected. This means

- all the preprocessor directives (staring with '#') will be moved into the first column. This is necessary to conform with K&R compilers.

- all the other lines will be indented according to the given blocks in C, i.e. it is controlled by the use of '{' and '}' characters.
- all TAB characters ('\t') will be removed.
- all the SPACE (' ') characters at the end of lines will be removed.

Caution!

The indent offered by here does not offer the same functionality as indent known from UNIX!

- Command line:
  `taexutil INDENT <inputfile> <outputfile>`
  `<inputfile>` and `<outputfile>` can be equal but both must be given.

Preprocessor

Note:

The main intention of the preprocessor given here is **not** to get an easy to read source file. Instead it is to get a preprocessed file that can be compiled by a target compiler (e.g. Keil, Tasking,...) because long macros are not supported by some target compilers.

The preprocessor utility should only be used to preprocess generated C files. The following tasks will be processed:

1. Copy the SDL to C compilers main header file (`ml_typ.h`/`sct-types.h`) beside the output file.
2. Put all the lines of the main header file containing `#include <...>` statements into comments
3. Preprocess the selected generated C file by using the default compiler (see chapter 3, *The Preference Manager*).
4. Remove all the empty lines in the preprocessed file.
5. Run Indent on the preprocessed file.
6. Remove all the comments surrounding `#include < ...>` in the preprocessed file.
7. Remove the private copy of the main header file.

Note:

All the files given with `#include "..."` will be included during the preprocessing.

- Command line:
  `taexutil PREPRO <inputfile> <outputfile> "<cmd>" <compiler_flag>`
  - `<inputfile>` and `<outputfile>` can be equal but both must be given.
  - `<cmd>` is the command line needed to invoke the preprocessor of the default compiler
  - `<compiler_flag>` is the define to select the target compiler. (Please see "Compiler Flag" on page 2936)

FAQs

- **How can I set the target directory in the Targeting Expert?**

  It is not possible to set the target directory in the Targeting Expert. The one given in the Organizer will be used instead. To make it possible to have different integrations in one target directory the Targeting Expert creates a Target Sub-Directory Structure.

- **Where can I add my own macro definitions?**

  There is a manual configuration file written for each integration. It is called `sct_mcf.h` for Cadvanced, `ml_mcf.h` for Cmicro and `extreme_user_cfg.h` for Cextreme.

  This header file is included in the SDL to C compiler's library during compilation and contains a section where you can insert your own macro definitions. The section begins with

  ```
  /* BEGIN User Code */
  ```

  and ends with the line

  ```
  /*   END User Code */
  ```

  All the text given between both lines will be saved if the manual configuration file is re-generated.

> **Caution!**
>
> The manual configuration file `sct_mcf.h` for Cadvanced is included only if the flag `USER_CONFIG` is set. This can be done in most cases by defining `-DUSER_CONFIG` in the compiler options.

- **How can I modify the value of `$sdtdir` in the generated makefile?**

  Per default the value of `$sdtdir` is set to `<installationdir>/sdt/sdtdir/<platformsdtdir>`. In some cases it is probably necessary to get the library files from another directory. In this case you can modify the *Library Directory* in the Make dialog. The sub-directory structure must be the same as given under `<installationdir>/sdt/sdtdir/<platformsdtdir>`.

- **How can I modify the configuration flags that are disabled in the Targeting Expert user interface?**

The Targeting Expert can be customized in a way so that the configuration flags listed under Configuration Settings to Be Set or Configuration Settings to Be Reset are not disabled. To achieve that open the Customize dialog via the menu choice *Tools* > *Customize* and select the Advanced Mode. (Please see "Customization" on page 2924 for more details)

- **How can I create my own pre-defined integration?**

  You can create your own pre-defined integrations by

  – selecting one of the distributed pre-defined integrations (the one that fits most to your needs
  – configure it as you like (maybe it is necessary to switch into the Advanced Mode to have access to all the dialogs)
  – export the settings like it is described in "Export" on page 2922#

  After you have done so the pre-defined integration will be available.

- **I have exported an pre-defined integration but my colleagues are not able to access it in my home directory/in my PC installation. What to do?**

  When exporting settings to pre-defined integration settings it is possible to

  – export as project settings - or
  – export as user settings

  You have probably exported as user settings, i.e. the *.its* file is not accessible for other users. You can move the `<integration>.its` file in the same directory where the `<systemname>.sdt` file of your project is saved. So it will be visible for all users working on this system.

Chapter

60

SDL C Compiler Driver (SCCD)

The SDL C Compiler Driver (SCCD) is a utility intended to simplify C level debugging of code generated by the SDL Suite and hand-written code. It can be invoked from the makefiles used by the SDL Suite or stand-alone from the OS command line.

Introduction

The SDL C Compiler Driver (SCCD) is intended to simplify C level debugging by generating an intermediate C file placed in a user defined directory. This C file has all its macros expanded and is optionally "beautified" (pretty-printed). For ease of use SCCD is used as a C compiler driver called from the SDL Suite generated or handwritten makefiles. SCCD may also be used as a command line option when compiling C code.

The simplest way to introduce this facility for use in the standard SDL Suite environment is to modify the `makeoptions` or `make.opt` file associated with the predefined run-time library to be used. To enable this feature in a modified the SDL Suite run-time library, the directory and file structure must be similar to the pre-defined one.

The `makeoptions` or `make.opt` file is found in the `$sdtdir/SCT[Kernel name]` directory. The only modification **required** is to change the line

```
sctCC = cc      /* or some other compiler executable name */
```

to

```
sctCC = sccd cc     /* or some other compiler executable name */
```

SCCD must of course be visible in your path.

To customize the behavior of SCCD, modify the configuration file `sccd_<your_compiler_type>.cfg` variables described in the following sections.

Syntax for Invoking

The following syntax can be used to invoke SCCD:

To execute SCCD:

```
sccd <C compiler command line>
```

The C compiler command line should be the normal command line used to compile a C file: it should include the name of your C compiler, possible compiler options, and finally the name of the C file.

To print the values of the variables in the configuration file `sccd.cfg`:

```
sccd
```

To print the variables with extra help information:

```
sccd -h
```

Return Codes

0: Success.

1: After printing “help”.

2: No `.c` file.

3: Could not open `InFile.c`.

4: Could not open `InFile.i` for write.

5: `sccdMOVE` or `sccdOUTFILEREDIR` needs to be defined.

6: Could not open `InFile.i` for read.

7: Could not open/create `TmpDir/InFile.c`.

Actions Performed by SCCD

1. Execute an optional first-user defined command (`sccdUSER_CMD1`).
2. Create a sub-directory for temporary files and the pre-processed `.c` files.
3. Execute an optional second user-defined command (`sccdUSER_CMD2`).
4. Run a C pre-processor pass to expand all macros.
5. Execute an optional third user-defined command (`sccdUSER_CMD3`).
6. Pretty print the file.
7. Optional clean-up of the sub-directory.
8. Optionally copy `.hs` files to the sub-directory.
9. Execute an optional fourth user-defined command (`sccdUSER_CMD4`). You should use this command if you wish to invoke the "indent" utility from SCCD (see "C Beautifier" on page 3021 for more information).
10. Optionally compile, i.e. run the original command line.
11. Optional clean-up of the sub-directory, but leave the pre-processed `.c` file(s) for debugging purposes.

Configuration File

The behavior of SCCD is defined using a number of variables, each starting with "sccd". The variables are defined in a configuration file, `sccd_<your_compiler_type>.cfg`. Select the configuration file that corresponds to your C compiler and copy this file as `sccd.cfg`. If run from the SDL Suite, SCCD uses `$sctdir/sccd.cfg` as the configuration file; otherwise SCCD searches for `sccd.cfg` in the current directory, `$SCCD`, and **(on UNIX)** `$HOME`.

> **Note:**
>
> If `sccd.cfg` is not found, hard-coded defaults suitable for the Gnu C compiler (gcc) are used.

Below are the variables that control the behavior of SCCD. Note that all characters in variable values are significant, including white spaces.

`sccdNAME = "Default"`

Compiler name as defined in `scttypes.h`.

`sccdINFILESUFFIX = ".c"`

The expected file name suffix of the In-file(s), Default ".c".

`sccdCPP = ""`

The name of the C pre-processor. If this is left empty, CPP is used. Default "".

`sccdCPPFLAGS = ""`

Enable CPP and do not remove comments. This is C compiler dependent. Default for gcc is "-P -E -C", and for cc "-C -P".

`sccdMACROPREFIX = "-D"`

CPP command line define MACRO prefix. Default "-D".

`sccdINCLUDE1 = "-I"`

CPP command line include-path prefix. Default "-I".

`sccdINCLUDE2 = ""`

Alternative CPP command line include-path prefix. Default "".

`sccdOUTFILEREDIR = "-o "`

Character sequence to control CPP output file name. If empty, use `sccdFMOVE` instead.

`sccdFMOVE = ""`

OS forced file move or copy command. Used instead of `sccdOUTFILEREDIR`. Default: "".

`sccdDELETE = "rm -f"`

OS forced delete file command. Default: "rm -f".

`sccdCOPY = "cp"`

OS normal copy command. Default: "cp".

`sccdCOMPILE = "ON"`

Controls whether the final compilation pass should be run or not. Values are: "OFF" and default is "ON".

`sccdDEBUG = "OFF"`

Enable execution. Values are: “ON” and default is “OFF”.

`sccdPURGE = "ON"`

Purge temporary files. Values are: “OFF” and default is “ON”.

`sccdUSE_HS = "OFF"`

When set “ON”, the `.hs` files are not included until the compilation pass. Values are: “ON” and default is “OFF”.

`sccdSILENT = "OFF"`

Enable trace printout. Values are: “ON” and default is “OFF”.

`sccdTMPDIR = "sccdtmp"`

Temporary directory for the pre-processing. Default is `sccdtmp`. Setting `sccdTMPDIR` to “” or “.” in the configuration file suppresses temporary directory creation.

```
sccdUSER_CMD1 = ""
sccdUSER_CMD2 = ""
sccdUSER_CMD3 = ""
sccdUSER_CMD4 = ""
```

User-defined commands (see “Actions Performed by SCCD” on page 3018). The following pseudo variables can be used in all but the first one (`sccdUSER_CMD1`):

- `%f` expands to In-file name without extension.
- `%p` expands to In-file path.
- `%d` expands to the value of `sccdTMPDIR`.

Example: `echo \"Pre-processed C-file = %p/%d/%f.c\"`

To include ‘#’ in `sccdUSER_CMDx` and `sccdTMPDIR`, enter **`\#`**
To include ‘"’ in `sccdUSER_CMDx` and `sccdTMPDIR`, enter **`\"`**
To include ‘\’ in `sccdUSER_CMDx` and `sccdTMPDIR`, enter **`\\`**

C Beautifier

If you need a C beautifier to further format the C code from the SDL Suite, do as follows.

Try the `indent` utility (courtesy of Joseph Arcaneaux), see "Additional required tools and utilities" on page 3 in chapter 1, *Platforms and Products* for information how to obtain it.

The indent executable must be placed in your path.

In order to easily invoke indent from SCCD, insert the following statement in `sccdUSER_CMD4` in the appropriate `sccd.cfg` file(s), assuming no other changes have been made to the `sccd.cfg` file(s):

For UNIX compiler environments:

```
sccdUSER_CMD4 = "indent -kr -l70 -i2 %p/%d/%f.c"
```

For "DOS"-like compiler environments:

```
sccdUSER_CMD4 = "indent -kr -l70 -i2 %p\\%d\\%f.c"
```

This setup gives a `.c` source file formatted according to rules very like the ones used in the Kernighan & Richie "C" book. It will also try to force lines to be shorter than 70 characters and will use 2 positions indentation in if/while/... statements.

A slightly more elaborate example:

```
sccdUSER_CMD4 = "indent -kr -l70 -br -nce -nlp -ci3
-i2 %p/%d/%f.c"
```

Chapter 60 SDL C Compiler Driver (SCCD)

Chapter

61 *The Master Library*

This chapter covers the following topics:

- **The structure of the source code of the SDL Suite Master Library**
- **The runtime model used for the programs generated by the SDL to C compiler. The chapter also describes the data structures for representing the various SDL objects.**
- **The memory requirements for applications**
- **How to make a customized Master Library**
- **The compilation switches which affect the properties of the Master Library**

Note that the Master Library is only used together with the Cadvanced/Cbasic SDL to C Compiler. It cannot be used together with the Cmicro SDL to C Compiler.

Introduction

This chapter describes the source code of the runtime library for applications generated by the Cadvanced/Cbasic SDL to C Compiler. Applications generated by the Cmicro SDL to C Compiler are **not** covered.

The chapter covers basically two topics:

1. The sections "File Structure" on page 3025, "The Symbol Table" on page 3028, and "The SDL Model" on page 3070 describe the runtime model for programs generated by the SDL Cadvanced/Cbasic SDL to C Compiler.

   Mainly it is the data structure used to represent various SDL objects that is discussed, both from the static point of view (the type definitions), and from the dynamic point of view (what information it represents and how it is initialized, changed, and used).

   The full runtime model that is used during simulations (with the monitor) is described. From this model, an optimization is made to obtain an application (not using the monitor). The optimization is discussed under "Compilation Switches" on page 3119.

2. In the sections "Compilation Switches" on page 3119, "Creating a New Library" on page 3139, and "Adaptation to Compilers" on page 3147, different aspects on how to make new versions of the runtime library are discussed.

   The compilation switches treated in the section "Compilation Switches" on page 3119 are used to determine the properties of the runtime library and the generated C code, while section "Creating a New Library" on page 3139 shows how to make new versions of the runtime library using for example new combinations of compilation switches.

   In the section "Adaptation to Compilers" on page 3147, porting issues are discussed.

File Structure

The runtime library is structured into a number of files. These are:

- `scttypes.h`
- `sctlocal.h`
- `sctpred.h`
- `sctsdl.c`
- `sctpred.c`
- `sctutil.c`
- `sctmon.c`
- `sctpost.c`
- `sctos.c`
- `post.h, sdt.h, itex.h, dll.h`

On UNIX:

- `post.o (use post64.o for solaris 64-bit)`

In Windows:

- `libpost.lib` (the statically linked library)
- `post.lib` (for dynamically linking)
- `post.dll` (the dynamically linked library)

On UNIX, all files can be found in the directory `$telelogic/sdt/sdt-dir/<machine dependent dir>/INCLUDE` where <*machine dependent dir*> is for example `sunos5sdtdir` on SunOS 5.

In Windows, all files can be found in the directory `<installation directory>\sdt\sdtdir\wini386\include`.

Description of Files

scttypes.h

This file contains type definitions and extern declarations of variables and functions. The file is included by `sctsdl.c, sctpred.c, sctutil.c, sctmon.c, sctpost.c, sctos.c`, and by each generated C file.

sctlocal.h

This file contains type definitions and extern declarations of variables and functions that are used only in the kernel. This file is not included in generated code.

sctpred.h

This file contains type definitions and extern declarations handling the predefined data types in SDL (except PId, which is in `scttypes.h`). This file is included in generated code via `scttypes.h`.

sctsdt.c

In this file the implementation of the SDL operations can be found, together with the functions used for scheduling. In more detail, this file contains groups of functions for:

- Handling and reporting SDL dynamic errors
- SDL operations, such as Output, Create, Stop, Nextstate, Set, Reset, together with help functions for these activities
- Initialization and the main loop (the scheduler).

sctpred.c

The functions implementing the operations defined in the SDL predefined data types can be fund in this file. Operators for PId is implemented in `sctsdl.c`.

sctutil.c

This file contains basic read and write functions together with functions to handle reading and writing of values of abstract data types, including the predefined data types. It also contains the functions for MSC trace.

sctmon.c

The `sctmon.c` file contains the functions that implement the monitor interface, that is, interpreting and executing monitor commands.

sctpost.c

This file contains all the basic functions that are used to connect a simulator with the other parts of the SDL Suite.

sctos.c

In this file, some functions that represent the dependencies of hardware, operating system and compiler are placed.

The basic functions necessary for an application are a function to read the clock and a function to allocate memory.

To move a generated C program plus the runtime library to a new platform (including a new compiler), the major changes are to be made in this file, together with writing a new section in `scttypes.h` to describe the properties of the new compiler.

post.h and sdt.h

These files are included in `sctpost.c` if the communication mechanism with other the SDL Suite applications should be part of the actual object code version of the library. The file `post.h` contains the function interface, while `sdt.h` contains message definitions.

Caution!

Windows only: When linking with the PostMaster's dynamically linked libraries (`post.lib` and `post.dll`), the environment variable `USING_DLL` must be defined before including `post.h`. Example:

```
#define USING_DLL
#include "post.h"
#undef USING_DLL
```

post.o (post64.o for solaris 64-bit, post.lib in Windows)

This file contains the implementation of functions needed to send messages, via the Postmaster, to other tools in the SDL Suite.

The Symbol Table

The symbol table is used for storing information mainly about the static properties of the SDL system, such as the block structure, connections of channels and the valid input signal set for processes. Some dynamic properties are also placed in the symbol table; for example the list of all active process instances of a process instance set. This is part of the node representing the process instance set.

The nodes in the symbol table are structs with components initialized in the declaration. During the initialization of the application, in the `yInit` function in generated code, a tree is built up from these nodes.

Symbol Table Tree Structure

The symbol table is created in two steps:

1. First, symbol table nodes are declared as structs with components initialized in the declaration (in generated code).
2. Then, the `yInit` function (in generated code) updates some components in the nodes and builds a tree from the nodes. This operation is not needed in an application!

The following names can be used to refer to some of the nodes that are always present. These names are defined in `scttypes.h`.

```
xSymbolTableRoot
xEnvId
xSrtN_SDL_Bit
xSrtN_SDL_Bit_String
xSrtN_SDL_Boolean
xSrtN_SDL_Character
xSrtN_SDL_Charstring
xSrtN_SDL_Duration
xSrtN_SDL_IA5String
xSrtN_SDL_Integer
xSrtN_SDL_Natural
xSrtN_SDL_Null
xSrtN_SDL_NumericString
xSrtN_SDL_Object_Identifier
xSrtN_SDL_Octet
xSrtN_SDL_Octet_String
xSrtN_SDL_PId
xSrtN_SDL_PrintableString
xSrtN_SDL_Real
xSrtN_SDL_Time
xSrtN_SDL_VisibleString
```

xSymbolTableRoot is the root node in the symbol table tree. Below this node the system node is inserted. After the system node, there is a node representing the environment of the system (xEnvId). Then there is one node for each package referenced from the SDL system. This is true also for the package predefined containing the predefined data types. The nodes for the predefined data types, that are sons to the node for the package predefined, can be directly referenced by the names xSrtN_SDL_xxx, according to the list above.

Nodes in the symbol table are placed in the tree exactly according its place of declaration in SDL. A node that represent an item declared in a block is placed as a child to that block node, and so on. The hierarchy in the symbol table tree will directly reflect the block structure and declarations within the blocks and processes.

A small example can be found in Figure 560 on page 3107. The following node types will be present in the tree:

Node Type	Description
xSystemEC	Represent the system or the system instance.
xSystemTypeEC	Represents a system type.
xPackageEC	Represents a package.
xBlockEC	Represent blocks and block instances.
xBlockTypeEC	Represents a block type.
xBlockSubstEC	Represents a block substructure and can be found as a child of a block node.
xProcessEC	Represent processes and process instances. The environment process node is placed after the system node and is used to represent the environment of the system.
xProcessTypeEC	Represents a process type.
xServiceEC	Represents a service or service instance.
xServiceTypeEC	Represents a service type.
xProcedureEC	Represents a procedure.

Node Type	Description
xOperatorEC	Represents an operator diagram, i.e. an ADT operator with an implementation in SDL.
xCompoundStmtEC	Represents a compound statement containing variable declarations.
xSignalEC xTimerEC	Represents a signal or timer type.
xRPCSignalEC	Represents the implicit signals (pCALL, pREPLY) used to implement RPCs.
xSignalParEC	There will be one signal parameter node, as a child to a signal, timer, and RPC signal node, for each signal or timer parameter.
xStartUpSignalEC	Represents a start-up signal, that is, the signal sent to a newly created process containing the actual FPAR parameters. An xStartUpSignalEC node is always placed directly after the node for its process.
xSortEC xSyntypeEC	Represents a newtype or a syntype.
Struct Component Node (xVariableEC)	A sort node representing a struct has one struct component node as child for each struct component in the sort definition.
xLiteralEC	A sort node similar to an enum type has one literal node as child for each literal in literal list.
xStateEC	Represents a state and can be found as a child to process and procedure nodes.
xVariableEC xFormalParEC	Represents a variable (DCL) or a formal parameter (FPAR) and can be found as children of process and procedure nodes.
xChannelEC xSignalRouteEC xGate	Represents a channel, a signal route, or a gate.
xRemoteVarEC	Represents a remote variable definition.
xRemotePrdEC	Represents a remote procedure definition.

Node Type	Description
xSyntVariableEC	Represents implicit variables or components introduced by the Cadvanced/Cbasic SDL to C Compiler. Used only by the Explorer.
xSynonymEC	Represent synonyms. Used only by the Explorer.

The nodes (the struct variables) will in generated code be given names according to the following table:

```
ySysR_SystemName
        (system, system type, system instance)
yPacR_PackageName
yBloR_BlockName
        (block, block type, block instance)
yBSuR_SubstructureName
yPrsR_ProcessName
        (process, process type, process instance)
yPrdR_ProcedureName  (procedure, operator)
ySigR_SignalName
        (signal, timer, startup signal, RPC signal)
yChaR_ChannelName    (channel, signal route, gate)
yStaR_StateName
ySrtR_NewtypeName    (newtype, syntype)
yLitR_LiteralName
yVarR_VariableName
       (variable, formal parameter, signal
        parameter, struct component, synt.variable)
yReVR_RemoteVariable
yRePR_RemoteProcedure
```

In most cases it is of interest to refer to a symbol table node via a pointer. By taking the address of a variable according to the table above, i.e.

```
& yPrsR_Process1
```

such a reference is obtained. For backward compatibility, macros according to the following example is also generated for several of the entity classes:

```
#define yPrsN_ProcessName  (&yPrsR_ProcessName)
```

Types Representing the Symbol Table Nodes

The following type definitions, from the file `scttypes.h`, are used in connection with the symbol table:

```
typedef enum {
  xRemoteVarEC,
  xRemotePrdEC,
  xSignalrouteEC,
  xStateEC,
  xTimerEC,
  xFormalParEC,
  xLiteralEC,
  xVariableEC,
  xBlocksubstEC,
  xPackageEC,
  xProcedureEC,
  xOperatorEC,
  xProcessEC,
  xProcessTypeEC,
  xGateEC,
  xSignalEC,
  xSignalParEC,
  xStartUpSignalEC,
  xRPCSignalEC,
  xSortEC,
  xSyntypeEC,
  xSystemEC,
  xSystemTypeEC,
  xBlockEC,
  xBlockTypeEC,
  xChannelEC,
  xServiceEC,
  xServiceTypeEC,
  xCompoundStmtEC,
  xSyntVariableEC
  xMonitorCommandEC
}   xEntityClassType;


typedef enum {
  xPredef, xUserdef, xEnum,
  xStruct, xArray, xGArray, xCArray,
  xOwn, xORef, xRef, xString,
  xPowerSet, xGPowerSet, xBag, xInherits, xSyntype,
  xUnion, xUnionC, xChoice
}   xTypeOfSort;

typedef char  *xNameType;
```

```
typedef struct xIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
}  xIdRec;


                                /*BLOCKSUBSTRUCTURE*/
typedef struct xBlockSubstIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
}  xBlockSubstIdRec;

                                     /*LITERAL*/
typedef struct xLiteralIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   int               LiteralValue;
}  xLiteralIdRec;

                                     /*PACKAGE*/
typedef struct xPackageIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
#ifdef XIDNAMES
   xNameType         ModuleName;
#endif
}  xPackageIdRec;
```

```
                                        /*SYSTEM*/
typedef struct xSystemIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xIdNode          *Contents;
   xPrdIdNode       *VirtPrdList;
   xSystemIdNode     Super;
#ifdef XTRACE
   int               Trace_Default;
#endif
#ifdef XGRTRACE
   int               GRTrace;
#endif
#ifdef XMSCE
   int               MSCETrace;
#endif
}  xSystemIdRec;


                          /*CHANNEL,SIGNALROUTE,GATE*/
#ifndef XOPTCHAN
typedef struct xChannelIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xSignalIdNode    *SignalSet; /*Array*/
   xIdNode          *ToId;      /*Array*/
   xChannelIdNode    Reverse;
}  xChannelIdRec;   /* And xSignalRouteEC.*/
#endif


                                        /*BLOCK*/
typedef struct xBlockIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
```

```
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xBlockIdNode      Super;
   xIdNode          *Contents;
   xPrdIdNode       *VirtPrdList;
   xViewListRec     *ViewList;
   int               NumberOfInst;
#ifdef XTRACE
   int               Trace_Default;
#endif
#ifdef XGRTRACE
   int               GRTrace;
#endif
#ifdef XMSCE
   int               MSCETrace;
   int               GlobalInstanceId;
#endif
}  xBlockIdRec;


                                     /*PROCESS*/
typedef struct xPrsIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xStateIdNode     *StateList;
   xSignalIdNode    *SignalSet;
#ifndef XNOUSEOFSERVICE
   xIdNode          *Contents;
#endif
#ifndef XOPTCHAN
   xIdNode          *ToId; /*Array*/
#endif
   int               MaxNoOfInst;
#ifdef XNRINST
   int               NextNr;
   int               NoOfStaticInst;
#endif
   xPrsNode         *ActivePrsList;
   xptrint           VarSize;
#if defined(XPRSPRIO) || defined(XSIGPRSPRIO) || \
    defined(XPRSSIGPRIO)
   int               Prio;
#endif
   xPrsNode         *AvailPrsList;
#ifdef XTRACE
   int               Trace_Default;
```

```
#endif
#ifdef XGRTRACE
   int               GRTrace;
#endif
#ifdef XBREAKBEFORE
   char   *(*GRrefFunc) (int, xSymbolType *);
   int               MaxSymbolNumber;
   int               SignalSetLength;
#endif
#ifdef XMSCE
   int               MSCETrace;
#endif
#ifdef XCOVERAGE
   long int         *CoverageArray;
   long int          NoOfStartTransitions;
   long int          MaxQueueLength;
#endif
   void            (*PAD_Function) (xPrsNode);
   void            (*Free_Vars) (void *);
   xPrsIdNode        Super;
   xPrdIdNode       *VirtPrdList;
   xBlockIdNode      InBlockInst;
#ifdef XBREAKBEFORE
   char             *RefToDefinition;
#endif
}  xPrsIdRec;


#ifndef XNOUSEOFSERVICE
                                        /*SERVICE*/
typedef struct xSrvIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xStateIdNode     *StateList;
   xSignalIdNode    *SignalSet;
#ifndef XOPTCHAN
   xIdNode          *ToId;
#endif
   xptrint           VarSize;
#ifdef XBREAKBEFORE
   char   *(*GRrefFunc) (int, xSymbolType *);
   int               MaxSymbolNumber;
   int               SignalSetLength;
#endif
#ifdef XCOVERAGE
   long int         *CoverageArray;
   long int          NoOfStartTransitions;
#endif
```

```
   xSrvNode          *AvailSrvList;
   void             (*PAD_Function) (xPrsNode);
   void             (*Free_Vars) (void *);
   xSrvIdNode         Super;
   xPrdIdNode        *VirtPrdList;
}  xSrvIdRec;
#endif


                                        /*PROCEDURE*/
typedef struct xPrdIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode            First;
   xIdNode            Suc;
#endif
   xIdNode            Parent;
#ifdef XIDNAMES
   xNameType          Name;
#endif
   xStateIdNode      *StateList;
   xSignalIdNode     *SignalSet;
   xbool             (*Assoc_Function) (xPrsNode);
   void              (*Free_Vars) (void *);
   xptrint            VarSize;
   xPrdNode          *AvailPrdList;
#ifdef XBREAKBEFORE
   char            *(*GRrefFunc) (int, xSymbolType*);
   int                MaxSymbolNumber;
   int                SignalSetLength;
#endif
#ifdef XCOVERAGE
   long int          *CoverageArray;
#endif
   xPrdIdNode         Super;
   xPrdIdNode        *VirtPrdList;
}  xPrdIdRec;


typedef struct xRemotePrdIdStruct {
   xEntityClassType    EC;
#ifdef XSYMBTLINK
   xIdNode             First;
   xIdNode             Suc;
#endif
   xIdNode             Parent;
#ifdef XIDNAMES
   xNameType           Name;
#endif
   xRemotePrdListNode  RemoteList;
}  xRemotePrdIdRec;


                                    /* SIGNAL, TIMER */
typedef struct xSignalIdStruct {
```

```
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xptrint           VarSize;
   xSignalNode      *AvailSignalList;
   xbool            (*Equal_Timer) (void *, void *);
#ifdef XFREESIGNALFUNCS
   void             (*Free_Signal) (void *);
#endif
#ifdef XBREAKBEFORE
   char             *RefToDefinition;
#endif
#if defined(XSIGPRIO) || defined(XSIGPRSPRIO) ||
defined(XPRSSIGPRIO)
   int               Prio;
#endif
}  xSignalIdRec;  /* and xTimerEC, xStartUpSignalEC,
                     and xRPCSignalEC.*/


                                    /*STATE*/
typedef struct xStateIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   int               StateNumber;
   xInputAction     *SignalHandlArray;
   int              *InputRef;
   xInputAction     (*EnablCond_Function)
                          (XSIGTYPE, void *);
   void             (*ContSig_Function)
                   (void *, int *, xIdNode *, int *);
   int                StateProperties;
#ifdef XCOVERAGE
   long int         *CoverageArray;
#endif
   xStateIdNode      Super;
#ifdef XBREAKBEFORE
   char             *RefToDefinition;
#endif
}  xStateIdRec;
```

```
                                        /*SORT*/
typedef struct xSortIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
#ifdef XFREEFUNCS
   void            (*Free_Function) (void **);
#endif
#ifdef XTESTF
   xbool           (*Test_Function) (void *);
#endif
   xptrint           SortSize;
   xTypeOfSort       SortType;
   xSortIdNode       CompOrFatherSort;
   xSortIdNode       IndexSort;
   long int          LowestValue;
   long int          HighestValue;
   long int          yrecIndexOffset;
   long int          typeDataOffset;
}  xSortIdRec;

                                        /*VARIABLE,...*/
typedef struct xVarIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xSortIdNode       SortNode;
   xptrint           Offset;
   xptrint           Offset2;
   int               IsAddress;
}  xVarIdRec;    /* And xFormalParEC and
                       xSignalParEC.*/


typedef struct xRemoteVarIdStruct {
   xEntityClassType     EC;
#ifdef XSYMBTLINK
   xIdNode              First;
   xIdNode              Suc;
#endif
   xIdNode              Parent;
#ifdef XIDNAMES
   xNameType            Name;
```

```
#endif
   xptrint              SortSize;
   xRemoteVarListNode   RemoteList;
}  xRemoteVarIdRec;
```

There are also pointer types defined for each of the `xECIdStruct` according to the following example:

```
typedef XCONST struct xIdStruct  *xIdNode;
```

The type definitions above define the contents in the symbol table nodes. Each `xECIdStruct`, where EC should be replaced by an appropriate string, have the first five components in common. These components are used to build the symbol table tree. To access these components, a pointer to a symbol table node can be type cast to any of the `xIdECNode` types. The type `xIdNode` is used as such general type, for example when traversing the tree.

The five components present in all `xIdNode` are:

- **`EC`** of type `xEntityClassType`. This component is used to determine what sort of SDL object the node represents. `xEntityClassType` is an enum type containing elements for all entity classes in SDL.
- **`First`**, **`Suc`**, and **`Parent`** of type `xIdNode`. These components are used to build the symbol table tree. `First` refers to the first child of the current node. `Suc` refers to the next brother, while `Parent` refers to the father node. Only `Parent` is needed in an application.
- **`Name`** of type `xNameType`, which is defined as `char *`. This component is used to represent the name of the current SDL object as a character string. Not needed in an application.

Next there are components depending on what entity class that is to be represented. Below we discuss the non-common elements in the other `xECIdStruct`.

Package

- **`ModuleName`** of type `xNameType`. If the package is generated from ASN.1, this component holds the name of the ASN.1 module as a `char *`.

System, System Type

- **Content** of type xIdNode *. This component contains a list of all channels at the system level.
- **VirtPrdList** of type xPrdIdNode *. This is a list of all virtual procedures in this system instance.
- **Super** of type xSystemIdNode. This is a reference to the inherited system type. In a system this component in null. In a system instance it is a reference to the instantiated system type.
- **Trace_Default** of type int. This component contains the current trace value defined for the system.
- **GRTrace** of type int. This component contains the current GR (graphical) trace value defined for the system.
- **MSCETrace** of type int. This component contains the current MSCE (Message Sequence Chart Editor) trace value defined for the system.

Channel, Signal route, Gate

For channels, signal routes, and gates there are always two consecutive xChannelIdNodes in the symbol table, representing the two possible directions for a channel, signal route, or gate. The components are:

- **SignalSet** of type xIdNode *. This component represents the signal set of the channel in the current direction (a unidirectional channel has an empty signal set in the opposite direction).

  SignalSet is an array with components referring to the xSignalIdNodes that represent the signals which are members of the signal set. The last component in the array is always a NULL pointer (the value (xSignalIdNode)0).

- **ToId** of type xIdNode *. This is an array of xIdNodes, where each array component is a pointer to a symbol table node representing an SDL object, which this Channel/Signal route/Gate is connected to (connected to in the sense: to the SDL objects that signals are sent forward to).

  The SDL objects that may be referenced in ToId are channels, signal routes, gates, processes, and services. The last component in the

array is always a NULL pointer (the value (xIdNode)0). See also "Channels and Signal Routes" on page 3106.

- **Reverse** of type xChannelIdNode. This is a reference to the symbol table node that represents the other direction of the same channel, signal route, or gate.

Block, Block Type, Block Instance

- **Super** of type xBlockIdNode. In a block, this component is NULL. In a block type this component is a reference to the block that this block inherits from (NULL if no inheritance). In a block instance, this is a reference to the block type that is instantiated.
- **Contents** of type xIdNode *. In a block instance, these components contains list of:
  - The process instantiations in the block
  - The signal routes in the block
  - The outgoing gates from the block
  - The processes in the block
  - The gates defined in process instantiations in the block.
- **VirtPrdList** of type xPrdIdNode *. This is a list of all virtual procedures in this block instance.
- **ViewList** of type xViewListRec *. This is a list of all revealed variables in the block or block instance.
- **NumberOfInst** of type int. This is the number of block instances in a block instance set. The component is thus only relevant for a block instance.
- **Trace_Default** of type int. This component contains the current value of the trace defined for the block.
- **GRTrace** of type int. This component contains the current value of the GR trace defined for the block.
- **MSCETrace** of type int. This component contains the current MSCE (Message Sequence Chart Editor) trace value defined for the block.
- **GlobalInstanceId** of type int. This component is used to store a unique id needed when performing MSCE trace.

Process, Process Type, Process Instance

- **StateList** of type `xStateIdNode *`. This is a list of references to the `xStateIdNodes` for this process or process type. Using the state value of an executing process, this list can be used to find the corresponding `xStateIdNode`.

- **SignalSet** of type `xIdNode *`. This represents the valid input signal set of the process or process type.

  `SignalSet` is an array with components that refer to `xSignalIdNodes` that represent the signals and timers which are part of the signal set. The last component in the array is always a `NULL` pointer (the value `(xSignalIdNode)0`).

- **Contents** of type `xIdNode *`. This is an array containing references to the `xSrvIdNodes` of the services and service instances in this process.

- **ToId** of type `xIdNode *`. This is an array of `xIdNode`, where each array component is a pointer to an IdNode representing an SDL object that this process or process instance is connected to (connected to in the sense: to the SDL objects that signals are sent forward to).

  The SDL objects that may be referenced in `ToId` are channels, signal routes, gates, processes, and services. The last component in the array is always a `NULL` pointer (the value `(xIdNode)0`). See also section "Channels and Signal Routes" on page 3106.

- **MaxNoOfInst** of type `int`. This represents the maximum number of concurrent processes that may exist according to the specification for the current process or process instance. An infinite number of concurrent processes is represented by -1.

- **NextNo** of type `int`. This is the instance number that will be assigned to the next instance that is created of this process instance set.

- **NoOfStaticInst** of type `int`.This component contains the number of static instance of this process instance set that should be present at start up. Used for process and process instance.

- **ActivePrsList** of type `xPrsNode *`. This is the address of a pointer to the "first" in the (single linked) list of active process instances of the current process or process instantiation.

The list is continued using the `NextPrs` component in the `xPrsRec` struct that is used to represent a process instance. The order in the list is such that the first created of the active process instances is last, and the latest created is first.

- **`VarSize`** of type `xptrint`. The size, in bytes, of the data area used to represent the process (the struct: `yVDef_`*`ProcessName`*).
- **`Prio`** of type `int`. This represents the process priority.
- **`AvailPrsList`** of type `xPrsNode`. This is the address to the avail list pointer for process instances that have stopped. The data area can later be reused in subsequent Create actions on this process or process instantiation.
- **`Trace_Default`** of type `int`. This component contains the current value of the trace defined for the process.
- **`GRTrace`** of type `int`. This component contains the current value of the GR trace defined for the process.
- **`GRrefFunc`**, which is a pointer to a function that, given a symbol number (number assigned to a process symbol), will return a string containing the SDT reference to that symbol.
- **`MaxSymbolNumber`** of type `int`. This component is the number of symbols contained in the current process or process type.
- **`SignalSetLength`** of type `int`. This component is the number of signals contained in the signal set of the current process or process type.
- **`MSCETrace`** of type `int`. This component contains the current MSCE (Message Sequence Chart Editor) trace value defined for the process.
- **`CoverageArray`** of type `long int *`. This component is used as an array over all symbols in the process. Each time a symbol is executed the corresponding array component is increased by 1.
- **`NoOfStartTransitions`** of type `long int`. This component is used to count the number of times the start transition of the current process is executed. This information is presented in the coverage tables.

- **MaxQueueLength** of type `long int`. This component is used to register the maximum input port length for any instance of the current process. The information is presented in the coverage tables.
- **PAD_Function**, which is a pointer to a function. This pointer refers to the `yPAD_`*`ProcessName`* function for the current process. This function is called when a process instance of this type is to execute a transition. The `PAD_Functions` will of course be part of generated code, as they contain the action defined in the process graphs.
- **Free_Vars**, which is a pointer to a function. This pointer refers to the `yFree_`*`ProcessName`* function for the current process. This function is called when the process performs a stop action to deallocate memory used by the local variables in the process.
- **Super** of type `xPrsIdNode`. In a process this component is `NULL`. In a process type this component is a reference to the process type that this process type inherits from (`NULL` if no inheritance). In a process instance set, this is a reference to the process type that is instantiated.
- **VirtPrdList** of type `xPrdIdNode *`. This is a list of all virtual procedures in this process instantiation.
- **InBlockInst** of type `xBlockIdNode`. This component is a reference to the block instance set (if any) that this process or process instantiation is part of.
- **RefToDefinition** of type `char *`. This is the SDT reference to this process.

Service, Service Type, Service Instance

- **StateList** of type `xStateIdNode *`. This is a list of the references to the `xStateIdNodes` for this service or service type. Using the state value of an executing service, this list can be used to find the corresponding `xStateIdNode`.
- **SignalSet** of type `xIdNode *`. This represents the valid input signal set of the service or service type.

  `SignalSet` is an array with components that refer to `xSignalIdNodes` that represent the signals and timers which are part of the signal set. The last component in the array is always a `NULL` pointer (the value `(xSignalIdNode)0`).

- **ToId** of type `xIdNode *`. This is an array of `xIdNode`, where each array component is a pointer to an IdNode representing an SDL object that this service or service instance is connected to (connected to in the sense: to the SDL objects that signals are sent forward to).

  The SDL objects that may be referenced in `ToId` are channels, signal routes, gates, processes, and service. The last component in the array is always a `NULL` pointer (the value `(xIdNode)0)`. See also section "Channels and Signal Routes" on page 3106.

- **VarSize** of type `xptrint`. The size, in bytes, of the data area used to represent the service (the struct: `yVDef_`*`ServiceName`*).

- **GRrefFunc**, which is a pointer to a function that, given a symbol number (number assigned to a service symbol), will return a string containing the SDT reference to that symbol.

- **MaxSymbolNumber** of type `int`. This component is the number of symbols contained in the current service or service type.

- **SignalSetLength** of type `int`. This component is the number of signals contained in the signal set of the current service or service type.

- **CoverageArray** of type `long int`. This component is used as an array over all symbols in the service. Each time a symbol is executed the corresponding array component is increased by 1.

- **NoOfStartTransitions** of type `long int`. This component is used to count the number of times the start transition of the current service is executed. This information is presented in the coverage tables.

- **AvailSrvList** of type `xSrvNode`. This is the address to the avail list pointer for service instances that have stopped. The data area can later be reused.

- **PAD_Function**, which is a pointer to a function. This pointer refers to the `yPAD_`*`ServiceName`* function for the current service. This function is called when a service instance of this type is to execute a transition. The `PAD_Functions` will of course be part of generated code, as they contain the action defined in the service graphs.

- **Free_Vars**, which is a pointer to a function. This pointer refers to the `yFree_`*`SeriveName`* function for the current service. This func-

tion is called when the service performs a stop action to deallocate memory used by the local variables in the service.

- **Super** of type xSrvIdNode. In a service this component is NULL. In a service type this component is a reference to the service type that this service type inherits from (NULL if no inheritance). In a service instantiation this is a reference to the service type that is instantiated.
- **VirtPrdList** of type xPrdIdNode *. This is a list of all virtual procedures in this service instantiation.

Procedure, Operator Diagram, Compound Statement

Note that operator diagrams and compound statements containing variable declarations are treated as procedures. However, such objects can, for example, not contain states.

- **StateList** of type xStateIdNode *. This is a list of references to the xStateIdNodes for this process or process type. Using the state value of an executing process, this list can be used to find the corresponding xStateIdNode.
- **SignalSet** of type xIdNode *. This represents the valid input signal set of the process or process type.

  SignalSet is an array with components that refer to xSignalIdNodes that represent the signals and timers which are part of the signal set. The last component in the array is always a NULL pointer (the value (xSignalIdNode)0).
- **Assoc_Function**, which is a pointer to a function. This pointer refers to the *yProcedureName* function for the current procedure. This function is called when the SDL procedure is called and will execute the appropriate actions. The *yProcedureName* functions will, of course, be part of generated code as they contain the action defined in the procedure graphs.
- **Free_Vars**, which is a pointer to a function. This pointer refers to the yFree_*ProcedureName* function for the current procedure. This function is called when the procedure performs a return action to deallocate memory used by the local variables in the procedure.
- **VarSize** of type xptrint. The size, in bytes, of the data area used to represent the procedure (struct yVDef_*ProcedureName*).

- **AvailPrdList** of type `xPrdNode *`. This is the address of the avail list pointer for the data areas used to represent procedure instances. At a return action the data area is placed in the avail list and can later be reused in subsequent Calls of this procedure type.
- **GRrefFunc**, which is a pointer to a function that given a symbol number (number assigned to a procedure symbol) will return a string containing the SDT reference to that symbol.
- **MaxSymbolNumber** of type `int`. This component is the number of symbols contained in the current procedure.
- **SignalSetLength** of type `int`. This component is the number of signals contained in the signal set of the current procedure.
- **CoverageArray** of type `long int`. This component is used as an array over all symbols in the procedure. Each time a symbol is executed the corresponding array component is increased by 1.
- **Super** of type `xPrdIdNode`. This component is a reference to the procedure that this procedure inherits from (`NULL` if no inheritance).
- **VirtPrdList** of type `xPrdIdNode *`. This is a list of all virtual procedures in this procedure.

Remote Procedure

- **RemoteList** of type `xRemotePrdListNode`. This component is the start of a list of all processes that exports this procedure. This list is a linked list of `xRemotePrdListStructs`, where each node contains a reference to the exporting process.

Signal, Timer, StartUpSignal, and RPC Signals

- **VarSize** of type `xptrint`. The size, in bytes, of the data area used to represent the signal (the struct: `yPDef_`*`SignalName`*).
- **AvailSignalList** of type `xSignalNode *`. This is the address to the avail list pointer for signal instances of this signal type.
- **Equal_Timer**, which is a pointer to a function. This pointer only refers to a function when this node is used to represent a timer with parameters.

  In this case the referenced function can be used to investigate if the parameters of two timers are equal or not, which is necessary at reset

actions. The `Equal_Timer` functions will be part of generated code. These functions are called from the functions `xRemoveTimer` and `xRemoveTimerSignal`, both defined in `sctsdl.c`

- **`Free_Signal`**, which is a function. This function takes a signal reference and returns any dynamic data referenced from the signal parameters to the pool of available memory.
- **`RefToDefinition`** of type `char *`. The SDT reference to the definition of the signal or timer.
- **Prio**, of type int. The priority of the signal.

State

- **`StateNumber`** of type `int`. The int value used to represent this state.
- **`SignalHandlArray`** of type `xInputAction` *. This component refers to an array of `xInputAction`, where `xInputAction` is an enum type with the possible values `xDiscard`, `xInput`, `xSave`, `xEnablCond`, `xPrioInput`.

  The array will have the same number of components as the `SignalSet` array in the node representing the process in which this state is contained. Each position in the `SignalHandlArray` represents the way the signal in the corresponding position in the `SignalSet` array in the process should be treated in this state.

  The last component in the `SignalHandlArray` is equal to `xDiscard`, which corresponds to the `0` value last in the `SignalSet`.

  If the `SignalHandlArray` contains the value `xInput`, `xSave`, or `xDiscard` at a given index, the way to handle the signal is obvious. If the `SignalHandlArray` contains the value `xEnablCond`, it is, however, necessary to calculate the enabling condition expression to know if the signal should cause an input or should be saved. This calculation is exactly the purpose of the `EnablCond_Function` described below.
- **`InputRef`** of type `int *`. This component is an array. If the `SignalHandlArray` contains `xInput`, `xPrioInput`, or `xEnablCond` at a certain index, this `InputRef` contains the symbol number for the corresponding input symbol in the graph.

- **EnablCond_Function**, which is a function that returns xInputAction. If the state contains any enabling conditions, this pointer will refer to a function. Otherwise it refers to 0. An EnablCond_Function takes a reference to an xSignalIdNode (referring to a signal) and a reference to a process instance and calculates the enabling condition for the input of the current signal in the current state of the given process instance.

  The function returns either of the values xInput or xSave. The EnablCond_Functions will of course be part of generated code, as they contain enabling condition expressions. These functions are called from the function xFindInputAction in the file sctsdl.c. xFindInputAction is used by the SDL_Output and SDL_Nextstate functions.

- **ContSig_Function**, which is a function returning int. If the state contains any continuous signals, this pointer will refer to a function. Otherwise it refers to 0.

- **StateProperties** of type int. In this component the three least significant bits are used to indicate:

  – If any enabling condition or continuous signal expression in the state contains a reference to an object that might change its value even though the process does not execute any actions.
  – If there are any priority inputs in the state.
  – If there are any virtual priority inputs in the state.

  Objects according to the first item in the list are: Now, Active (timer is active), Import, View, and Sender. StateProperties is used in the function SDL_Nextstate to take appropriate actions when a process enters a state.

- **CoverageArray** of type long int. This component is used as an array over the signalset (+1) of the process. Each time an input operation is performed, the corresponding array component is increased by 1. The last component, at index equal to the length of the signalset, is used to record the number of continuous signals “received” in the state. The information stored in this component is presented in the coverage table.

- **Super** of type xPrdIdNode. This component is a reference to the procedure that this procedure inherits from (NULL if no inheritance).

- **RefToDefinition** of type `char *`. The SDT reference to the definition of the state (one of the symbols where this state is defined).

Sort and Syntype

- **Free_Function**, which is a function. This function pointer is non-`0` for types represented using dynamic memory (Charstring, Octet_string, Strings, Bags, for example). The `Free_Functions` are used to return dynamic memory to the pool of dynamic memory.
- **Test_Function**, which is a function returning `xbool`. This function is non-0 for all types containing range conditions. The function pointers are used by the monitor system to check the validity of a value when assigning it to a variable.
- **SortSize** of type `xptrint`. This component represents the size, in bytes, of a variable of the current sort.
- **SortType** of type `xTypeOfSort`. This component indicates the type of sort. Possible values are: `xPredef`, `xUserdef`, `xEnum`, `xStruct`, `xArray`, `xGArray`, `xCArray`, `xRef`, `xString`, `xPowerSet`, `xBag`, `xGPowerSet`, `xInherits`, `xSyntype`, `xUnion`, `xUnionC`, and `xChoice`.

SortType is `xArray, xGArray, xCArray`

- **CompOrFatherSort** of type `xSortIdNode`. This is a pointer to the `SortIdNode` that represents the component sort.
- **IndexSort** of type `xSortIdNode`. This is a pointer to the `SortIdNode` that represents the index sort. In a xCArray the index sort in always Integer.
- In xGArray, **LowestValue** is used as the offset of `Data` in the `xxx_ystruct`.
  In xArray and xCArray it is 0.
- In xGArray, **HighestValue** is used as the size of the `xxx_ystruct`.
  In xArray it is 0.
  In xCArray it is the highest index, i.e. the Length - 1.
- In xGArray, **yrecIndexOffset** is used as the offset of `Index` in the `xxx_ystruct`.
  In xArray and xCArray it is 0.

- In xGArray, **yrecDataOffset** is used as the offset of `Data` in the type (i.e. the value representing the default value).
  In xArray and xCArray it is 0.

SortType is `xString, xGPowerSet, xBag`

- **`CompOrFatherSort`** of type `xSortIdNode`. This is a pointer to the `SortIdNode` that represents the component sort.
- **`LowestValue`** is used as the offset of `Data` in the `xxx_ystruct`.
- **`HighestValue`** is used as the size of the `xxx_ystruct`.

SortType is `xPowerSet, xRef, xOwn, xORef`

- **`CompOrFatherSort`** of type `xSortIdNode`. This is a pointer to the `SortIdNode` that represents the component sort.

SortType is `xInherits`

- **`CompOrFatherSort`**, of type `xSortIdNode`. This is a pointer to the `SortIdNode` that represents the inherited sort.

SortType is `xSyntype`

- **`CompOrFatherSort`,** of type `xSortIdNode`. This is a pointer to the `SortIdNode` that represents the father sort (the newtype from which the syntype originates, even if it is a syntype of a syntype).
- **`IndexSort`**, of type `xSortIdNode`. This is a pointer to the SortIdNode that represents the represents the father sort (the newtype or syntype from which the syntype originates).
- **`LowestValue`**, of type `long int`. If the syntype can be used as an index in an array (translated to a C array) then this value is the lowest value in the syntype range, otherwise it is 0.
- **`HighestValue`**, of type `long int`. If the syntype can be used as an index in an array (translated to a C array) then this value is the highest value in the syntype range, otherwise it is 0. The `LowestValue` and `HighestValue` are used by the monitor when it handles arrays with this type as index type.

Variable, FormalPar, SignalPar, and Struct Components

- **`SortNode`** of type `xSortIdNode`. This component is a pointer to the SortIdNode that represents the sort of this variable or parameter.
- **`Offset`** of type `xptrint`. This component represents the offset, in bytes, within the struct that represents the process or procedure variables, the signal parameter, or the SDL struct. In other words, this is the relative place of this component within the struct.
- **`Offset2`** of type `xptrint`. For a formal parameter in a process this component represents the offset, in bytes, of a formal parameter in the `StartUpSignal`. For an exported variable in a process this component represents the offset, in bytes, of the exported value for this variable.
- **`IsAddress`** of type `int`. This component is only used for procedure and operator formal parameters and is then used to indicate if the parameter in IN or IN/OUT or a result variable.

Remote Variable

- **`SortSize`** of type `xptrint`. This component is the size of the type of the exported variables.
- **`RemoteList`** of type `xRemoteVarListNode`. This component is the start of a list of all processes that exports this variable. This list is a linked list of `xRemoteVarListStruct`s, where each node contains a reference to the exporting process and the `Offset` where to find the exported value.

Type Info Nodes

This section describes the most important implementation details regarding the type info node. Type info nodes are data structures that are used during run-time by the functions providing generic implementations of SDL operators. As the type info nodes contain essentially the same information as the xSortIdNodes, the type info nodes are used in more and more places in the code where xSortIdNode previously were used. In the longer perspective the xSortIdNode will be removed completely.

The type definitions that describe the type info nodes are listed in the `sctpred.h` file.

Each type info node is a struct that consists of:

- general components that are available for all type info nodes
- type-specific components that describe each specific type.

The following utility macros can be used to configure the type info nodes:

```
#ifndef T_CONST
#define T_CONST const
#endif

#ifndef T_SDL_EXTRA_COMP
#define T_SDL_EXTRA_COMP
#define T_SDL_EXTRA_VALUE
#endif

#ifndef T_SDL_USERDEF_COMP
#define T_SDL_USERDEF_COMP
#endif

#if defined(XREADANDWRITEF) && !defined(T_SDL_NAMES)
#define T_SDL_NAMES
#endif

#ifdef T_SDL_NAMES
#define T_SDL_Names(P) , P
#else
#define T_SDL_Names(P)
#endif

#ifdef T_SIGNAL_SDL_NAMES
#define T_Signal_SDL_Names(P) , P
#else
#define T_Signal_SDL_Names(P)
#endif

#ifdef T_SDL_INFO
#define T_SDL_Info(P) , P
#else
#define T_SDL_Info(P)
#endif

#ifndef XNOUSE_OPFUNCS
#define T_SDL_OPFUNCS(P) , P
#else
#define T_SDL_OPFUNCS(P)
#endif

struct tSDLFuncInfo;
```

General Components

The following components are available for all type info nodes. The definition of the components is only listed in this section, but it is valid for each type info node listed in the next section.

```
/* --- General type information for SDL types --- */

typedef T_CONST struct tSDLTypeInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
} tSDLTypeInfo;
```

- **TypeClass**: This component defines which type the info node describes. A list of available types and their corresponding values can be found in the enum type definition below:

```
typedef enum
{
  /*SDL - standard types*/
  type_SDL_Integer=128,
  type_SDL_Real=129,
  type_SDL_Natural=130,
  type_SDL_Boolean=131,
  type_SDL_Character=132,
  type_SDL_Time=133,
  type_SDL_Duration=134,
  type_SDL_Pid=135,
  type_SDL_Charstring=136,
  type_SDL_Bit=137,
  type_SDL_Bit_string=138,
  type_SDL_Octet=139,
  type_SDL_Octet_string=140,
  type_SDL_IA5String=141,
  type_SDL_NumericString=142,
  type_SDL_PrintableString=143,
  type_SDL_VisibleString=144,
  type_SDL_NULL=145,
  type_SDL_Object_identifier=146,

  /* SDL - standard ctypes */
  type_SDL_ShortInt=150,
  type_SDL_LongInt=151,
```

```
    type_SDL_UnsignedShortInt=152,
    type_SDL_UnsignedInt=153,
    type_SDL_UnsignedLongInt=154,
    type_SDL_Float=155,
    type_SDL_Charstar=156,
    type_SDL_Voidstar=157,
    type_SDL_Voidstarstar=158,

    /* SDL - user defined types */
    type_SDL_Syntype=170,
    type_SDL_Inherits=171,
    type_SDL_Enum=172,
    type_SDL_Struct=173,
    type_SDL_Union=174,
    type_SDL_UnionC=175,
    type_SDL_Choice=176,
    type_SDL_ChoicePresent=177,
    type_SDL_Powerset=178,
    type_SDL_GPowerset=179,
    type_SDL_Bag=180,
    type_SDL_String=181,
    type_SDL_LString=182,
    type_SDL_Array=183,
    type_SDL_Carray=184,
    type_SDL_GArray=185,
    type_SDL_Own=186,
    type_SDL_Oref=187,
    type_SDL_Ref=188,
    type_SDL_Userdef=189,
    type_SDL_EmptyType=190,

    /* SDL - signals */
    type_SDL_Signal=200,
    type_SDL_SignalId=201

} tSDLTypeClass;
```

- **OpNeeds**: This component contains four bits that give the properties of the type regarding assignment, equal test, free function, and initialization.

  – The first bit indicates if the type is a pointer that needs to be automatically freed, or if it contains a pointer that needs to be au-

tomatically freed. If the first bit is set, it is necessary to look for memory to be freed inside of a value of this type.
- The second bit indicates if memcmp can be used to test if two values of this type are equal or not. If the bit is set, special treatment is needed.
- The third bit indicates if memcpy can be used to perform assign of this type. If the bit is set, special treatment is needed.
- The fourth bit indicates if this type needs to be initialized to anything else than 0.

The following macros can be used to test these properties:

```
#define NEEDSFREE(P) \
  (((tSDLTypeInfo *)(P))->OpNeeds & (unsigned char)1)
#define NEEDSEQUAL(P) \
  (((tSDLTypeInfo *)(P))->OpNeeds & (unsigned char)2)
#define NEEDSASSIGN(P) \
  (((tSDLTypeInfo *)(P))->OpNeeds & (unsigned char)4)
#define NEEDSINIT(P) \
  (((tSDLTypeInfo *)(P))->OpNeeds & (unsigned char)8)
```

- **SortSize**: This component defines the size of the type.
- **OpFuncs**: This is a pointer to a struct containing references to specific assign, equal, free, read, and write functions. This component is only used in special cases. If assign, equal, free, read or write functions have been implemented using #ADT directives, information about this is stored in the OpFuncs field. The default value of the OpFuncs field is 0, but if you have provided any of these functions, the field will be a pointer to a tSDLFuncInfo struct. This struct will in turn refer to the provided functions.

```
typedef struct tSDLFuncInfo {
  void *(*AssFunc) (void *, void *, int);
  SDL_Boolean (*EqFunc) (void *, void *);
  void (*FreeFunc) (void **);
#ifdef XREADANDWRITEF
  char *(*WriteFunc) (void *);
  int (*ReadFunc) (void *);
#endif
} tSDLFuncInfo;
```

- **Name**: This is the name of the type as a string literal.
- **FatherScope**: This is a pointer to the IdNode for the scope that the type is defined in.
- **SortIdNode**: This is a pointer to the xSortIdNode that describes the same type. This field will in a longer perspective be removed.

Type Info Node Optimization

Depending on the operations used for data types in the translated system, some of the type info nodes might not be needed. The purpose of this described optimization is to remove such type info nodes.

The first step is to surround all type info nodes with #ifdefs according to the example for the predefined type integer below.

```
 #ifndef XTNOUSE_Integer
tSDLTypeInfo ySDL_SDL_Integer = {
  ....
  };
#endif
```

This means that the type info node for integer can be removed by defining XTNOUSE_Integer.

The names for the predefined data types in the `#ifndef` statements are:

```
#ifndef XTNOUSE_Integer
#ifndef XTNOUSE_Real
#ifndef XTNOUSE_Natural
#ifndef XTNOUSE_Boolean
#ifndef XTNOUSE_Character
#ifndef XTNOUSE_Time
#ifndef XTNOUSE_Duration
#ifndef XTNOUSE_Pid
#ifndef XTNOUSE_Charstring
#ifndef XTNOUSE_Bit
#ifndef XTNOUSE_Bit_string
#ifndef XTNOUSE_Octet
#ifndef XTNOUSE_Octet_string
#ifndef XTNOUSE_IA5String
#ifndef XTNOUSE_NumericString
#ifndef XTNOUSE_PrintableString
#ifndef XTNOUSE_VisibleString
#ifndef XTNOUSE_NULL
#ifndef XTNOUSE_Object_identifier
```

For user defined types the name is selected according to the following algorithm.

1. If the type name is unique (case sensitive, as C is case sensitive), that is there is only one data type in the system with this name, then name in the `#ifndef` will be:

```
XTNOUSE_typename
```

2. If the type name is not unique but type name plus the name of the scope where the type is defined is unique, the name in the `#ifndef` will be:

```
XTNOUSE_typename_scopename
```

3. In other cases the name in the `#ifndef` will be:

```
XTNOUSE_typename-with-prefix-or-suffix
```

Using only this part of the algorithm it is of course possible to manually write a `.h` file with the suitable defines to remove the type info nodes that are not used. The compiler/linker can help finding unused data.

However the code generator calculates the usage of type info nodes. This information will be stored in the file:

```
sdl_cfg.h
```

The last section in this file will contain the defines for the usage of type info nodes.

Example 509: Usage of type info nodes

```
 #ifdef XUSE_TYPEINFONODE_CFG
/* Type info node configuration */
#define XTNOUSE_Boolean
#define XTNOUSE_Character
#define XTNOUSE_Charstring
/* NOT #define XTNOUSE_Integer*/
....
....
/* NOT #define XTNOUSE_s*/
#endif
```

For every data type in the system there will be one line indicating if the type info node is used or not. Please note also that it is necessary to define XUSE_TYPEINFONODE_CFG at compilation, otherwise the automatically computed type info node optimization will not be used.

The file sdl_cfg.h is automatically included and used in Cmicro. In Cadvanced and C Code Generator the following code can be found in sctypes.h:

```
#ifdef USER_CONFIG
#include "sct_mcf.h"
#elif defined(AUTOMATIC_CONFIG)
#include "sdl_cfg.h"
#endif
```

So by defining USER_CONFIG and including sdl_cfg.h in sct_mcf.h or by defining AUTOMATIC_CONFIG the computed type info node optimization is used.

Sometimes the automatic computation on used type info nodes might fail. The most obvious case is usage inside inline C code. As the code generator does not parse such code, it has no chance to know of such usage. To cope with these situations the user has the possibility to tell the code generator that certain type info nodes are used, and thereby also the nodes that the particular node depends on. It is of course possible for a user to manually handle these situations be inserting proper #define and `#undef` after the type info node configuration. However this might prove difficult due to all the dependencies between type info nodes.

A user can tell the code generator that certain type info nodes are used by specifying this in a file. The code generator will look for such a file according to the following:

If the environment variable TAU_TYPEINFOCFG is defined, the value of this variable is treated as a file name (including path) and this file is read. If the code generator can not open this file it will produce an error.

If the environment variable is not defined a file with the name 'typeinfo.cfg' is looked for. In SDL Suite the code generator will first look in the source directory and if it can not find the file there it will look in the directory where SDL Suite is started from. If a 'typeinfo.cfg' file is found it is read. If no such file is found the code generator assumes that the user does not have a type info configuration file.

The contents of the type info configuration file should be:

Each type that should be registered as used should be mentioned on a line of its own, starting with the type name. If the several data types with the same name exist in the system the type name can be followed be the name of the scope that the data is defined in. One or more spaces or tabs should separate the type name and the scope name.

Example 510: Type name and scope name ----------

```
typename1
typename2 scopename2
typename3
```

The code generator will search for data types that match the criteria (name or name/scope) given above. The search will in SDL Suite follow the case sensitivity option in the tool. The following rules then apply:

- The type info node in all data types that matches the criteria (type name or type name/scope name) will be registered as used.
- All type info nodes that the registered node depends on will also be registered as used.
- An error message will be given if no data type matches a criteria.

If case sensitive search is used the predefined types should be given according to the following table:

```
Integer
Real
Natural
Boolean
Character
Time
Duration
Pid
Charstring
Bit
Bit_string
Octet
Octet_string
IA5String
NumericString
PrintableString
VisibleString
NULL
Object_identifier
```

The name of the scope for these types is: **Predefined**

Memory optimization

In the kernel file sctpred.c the following functions are conditionally defined (using #ifdef) to save memory when they are not needed:

```
GenericIsAssigned
IsPointerToMake
GenericMakeStruct
GenericGetValue
GenericSetValue
```

The conditional definitions are based on usage in the kernel or generated code. However if there is a need for some reason to use them directly, it is needed to make them accessible by manually defining a flag corresponding to each function named XMK_USE_FUNC_<function name in capital letters>:

```
XMK_USE_FUNC_GENERICISASSIGNED
XMK_USE_FUNC_ISPOINTERTOMAKE
```

```
XMK_USE_FUNC_GENERICMAKESTRUCT
XMK_USE_FUNC_GENERICGETVALUE
XMK_USE_FUNC_GENERICSETVALUE
```

Furthermore the function GenericDefault has additional #ifdef constructions inside the function to save memory in some cases.

Type-Specific Components

The following section lists the components that defines the type info nodes. Only the type-specific components are explained. The general components are listed and explained in the section above.

Enumeration types

```
/* ------------- Enumeration type --------------- */
typedef T_CONST struct {
  int             LiteralValue;
  char           *LiteralName;
} tSDLEnumLiteralInfo;

typedef T_CONST struct tSDLEnumInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
#ifdef XREADANDWRITEF
  int             NoOfLiterals;
  tSDLEnumLiteralInfo *LiteralList;
#endif
} tSDLEnumInfo;
```

- **NoOfLiterals**: The number of literals in the enum type.
- **LiteralList**: a pointer to an array of `tSDLEnumLiteralInfo` elements. This list implements a translation table between enum values and literal names as strings

Syntypes, types with inheritance, and Own, Ref, Oref instantiations

```
/* ------ Syntype, Inherits, Own, Oref, Ref ----- */
typedef T_CONST struct tSDLGenInfoS {
```

```
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLTypeInfo   *CompOrFatherSort;
} tSDLGenInfo;
```

- **CompOrFatherSort**: Reference to the type info node of the father sort (syntype, inherits) or component sort (Own, Ref, Oref).

Powersets (implemented as unsigned in [ ])

```
/* ------------------ Powerset ------------------ */
typedef T_CONST struct tSDLPowersetInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLTypeInfo   *CompSort;
  int             Length;
  int             LowestValue;
} tSDLPowersetInfo;
```

- **CompSort**: Reference to the type info node of the component sort.
- **Length**: The number of possible values in the component sort.
- **LowestValue**: The value of the lowest value in the component sort.

Structs

```
/* ------------------- Struct ------------------- */
typedef int (*tGetFunc) (void *);
typedef void (*tAssFunc) (void *, int);
```

```
typedef T_CONST struct {
  xptrint     OffsetPresent; /* 0 if not optional */
  void       *DefaultValue;
} tSDLFieldOptInfo;


typedef T_CONST struct {
  tGetFunc        GetTag;
  tAssFunc        AssTag;
} tSDLFieldBitFInfo;


typedef T_CONST struct {
  tSDLTypeInfo   *CompSort;
#ifdef T_SDL_NAMES
  char           *Name;
#endif
  xptrint         Offset;     /* ~0 for bitfield */
  tSDLFieldOptInfo *ExtraInfo;
} tSDLFieldInfo;


typedef T_CONST struct tSDLStructInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLFieldInfo  *Components;
  int             NumOfComponents;
} tSDLStructInfo;
```

- **Components**: An array of tSDLFieldInfo; one component in the array for each field of the struct.
- **NumOfComponents**: The number of fields in the struct.
- **CompSort in tSDLFieldInfo**: The reference to the type info node of the field sort.
- **Name in tSDLFieldInfo**: The name of the field as a string.

- **Offset in tSDLFieldInfo**: The offset of the field in the C struct that represents the SDL struct. This component is ~0 for bitfield in SDL (offsets cannot be calculated for bitfields).
- **ExtraInfo in tSDLFieldInfo**: The interpretation of this component depends on the properties in the SDL field.
  - if Offset is ~0, the field is a bitfield and ExtraInfo is a pointer to a tSDLFieldBitFInfo struct containing two functions to set and get the value of the bitfield.
  - if Offset is not ~0 and ExtraInfo != 0, the SDL field is either optional or has a default value. ExtraInfo is a pointer to a tSDLFieldOptInfo struct containing the offset for the Present flag (0 if not optional) and a pointer to the default value (0 if no default value).

Choice and #union

```
/* ---------------- Choice, Union --------------- */

typedef T_CONST struct {
  tSDLTypeInfo        *CompSort;
#ifdef T_SDL_NAMES
  char                *Name;
#endif
} tSDLChoiceFieldInfo;

typedef T_CONST struct tSDLChoiceInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLChoiceFieldInfo *Components;
  int                  NumOfComponents;
  xptrint              OffsetToUnion;
  xptrint              TagSortSize;
#ifdef XREADANDWRITEF
  tSDLTypeInfo        *TagSort;
#endif
} tSDLChoiceInfo;
```

- **Components**: An array of tSDLChoiceFieldInfo; one component in the array for each field in the choice/#union.
- **NumOfComponents**: The number of fields in the choice/#union.
- **OffsetToUnion**: The offset to where the union, within the representation of the choice/#union, starts.
- **TagSortSize**: The size of the tag type.
- **TagSort**: A reference to the type info node of the tag sort.

Array and Carray

```
/* ---------------- Array, CArray ---------------- */
typedef T_CONST struct tSDLArrayInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLTypeInfo   *CompSort;
  int             Length;
#ifdef XREADANDWRITEF
  tSDLTypeInfo   *IndexSort;
  int             LowestValue;
#endif
} tSDLArrayInfo;
```

- **CompSort**: The reference to the type info node of the component sort.
- **Length**: The number of components in the array.
- **IndexSort**: The reference to the type info node of the index sort.
- **LowestValue**: The start value of the index range (as an int).

General arrays

A general array is an array that is represented as a linked list in C.

```
/* ------------------- GArray ------------------- */
typedef T_CONST struct tSDLGArrayInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLTypeInfo   *IndexSort;
  tSDLTypeInfo   *CompSort;
  xptrint         yrecSize;
  xptrint         yrecIndexOffset;
  xptrint         yrecDataOffset;
  xptrint         arrayDataOffset;
} tSDLGArrayInfo;
```

- **IndexSort**: The reference to the type info node of the index sort.
- **CompSort**: The reference to the type info node of the component sort.
- **yrecSize**: The size of the type SDLType_yrec.
- **yrecIndexOffset**: The offset of Index in type SDLType_yrec.
- **yrecDataOffset**: The offset of Data in type SDLType_yrec.
- **arrayDataOffset**: The offset of Data in type SDLType, where SDLType is the name in C of the translated array type.

General powersets, Bags, Strings and Object_identifier

```
/* -- GPowerset, Bag, String, Object_Identifier - */
typedef T_CONST struct tSDLGenListInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
```

```
#endif
  tSDLTypeInfo   *CompSort;
  xptrint         yrecSize;
  xptrint         yrecDataOffset;
} tSDLGenListInfo;
```

- **CompSort**: The reference to the type info node of the component sort.
- **yrecSize**: The size of the type SDLType_yrec
- **yrecDataOffset**: The offset of Data in type SDLType_yrec

Limited strings

A limited string is a string that is implemented as an array in C.

```
/* ------------------ LString ------------------- */
typedef T_CONST struct tSDLLStringInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLTypeInfo   *CompSort;
  int             MaxLength;
  xptrint         DataOffset;
} tSDLLStringInfo;
```

- **CompSort**: The reference to the type info node of the component sort.
- **MaxLength**: The maximum length of the string.
- **DataOffset**: The offset of Data in type SDLType, where SDLType is the name in C of the translated string type.

SDL type (C representation decided with a #ADT directive)

```
/* ------------------ Userdef ------------------- */
/* used for user defined types #ADT(T(h)) */
typedef T_CONST struct tSDLUserdefInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
```

```
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  T_SDL_USERDEF_COMP
} tSDLUserdefInfo;
```

SDL signal

A signal is treated in the same way as a struct.

```
/* ------------------- Signal ------------------- */
typedef T_CONST struct {
  tSDLTypeInfo       *ParaSort;
  xptrint             Offset;
} tSDLSignalParaInfo;

typedef T_CONST struct tSDLSignalInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SIGNAL_SDL_NAMES
  char           *Name;
#endif
  tSDLSignalParaInfo *Param;
  int             NoOfPara;
} tSDLSignalInfo;
```

- **Param**: An array with a component of the type tSDLSignalParaInfo for each signal parameter type. For each parameter, the parameter sort is given as a reference to the type info node and as the offset for the parameter value within the struct representing the signal.
- **NoOfPara**: The number of parameters in the signal.

The SDL Model

Signals and Timers

Data Structure Representing Signals and Timers

A signal is represented by a struct type. The `xSignalRec` struct, defined in `scttypes.h`, is a struct containing general information about a signal except from the signal parameters. In `scttypes.h` the following information about signals can be found:

```
#ifdef XMSCE
#define GLOBALINSTID  int GlobalInstanceId;
#else
#define GLOBALINSTID
#endif

#if defined(XSIGPATH) && defined(XMSCE)
#define ENVCHANNEL  xChannelIdNode EnvChannel;
  /* Used if env split into channels in MSC trace */
#else
#define ENVCHANNEL
#endif

#ifdef XENV_CONFORM_2_3
#define XSIGNAL_VARP  void * VarP;
#else
#define XSIGNAL_VARP
#endif

define SIGNAL_VARS \
   xSignalNode   Pre; \
   xSignalNode   Suc; \
   int           Prio; \
   SDL_PId       Receiver; \
   SDL_PId       Sender; \
   xSignalIdNode NameNode; \
   GLOBALINSTID \
   ENVCHANNEL \
   XSIGNAL_VARP

typedef struct xSignalStruct  *xSignalNode;
typedef struct xSignalStruct {
   SIGNAL_VARS
}  xSignalRec;
```

The `xSignalNode` type is thus a pointer type which is used to refer to allocated data areas of type `xSignalRec`. The components in the `xSignalRec` struct are used as follows:

- **`Pre`** and **`Suc`**. These pointers are used to link a signal into the input port of the receiving process instance.

The input port is a doubly linked list of signals. `Suc` is also used to link a signal into the avail lists for the current signal type. This list can be found in the `SignalIdNode` that represents this signal type. If the signal is in the avail list `Pre` is `0`.

- **`Prio`** is used to represent the priority of the signal instance. Signal priorities are used by continuous signals and by ordinary signals if signal priorities are defined (signal priority is a possible extension provided in the product).
- **`Receiver`** is used to reference the receiver of the signal. It is either set in the output statement (OUTPUT TO), or calculated (OUTPUT without TO).
- **`Sender`** is the `PId` value of the sending process instance. This value is necessary to provide the SDL function SENDER.
- **`NameNode`** is a reference to the `xSignalIdNode` representing the signal type and thus defines the signal type of this signal instance.
- **`VarP`** is a pointer introduced via the macro `XSIGNAL_VARP` to make signal compatible with SDT 2.3. Normally this components is **not** present.
- **`EnvChannel`** is used to identify the outgoing channel in MSCE trace.
- **`GlobalInstanceId`** is used in the MSCE trace as a unique identification of the signal instance.

A signal without parameters are represented by a `xSignalStruct`, while for signals with parameters a struct type named `yPDef_`*`SignalName`* and a pointer type referencing this struct type (`yPDP_`*`SignalName`*) are defined in generated code. The struct type will start with the `SIGNAL_VARS` macro and then have one component for each signal parameter, in the same order as the signal parameters are defined. The components will be named Param1, Param2, and so on.

Example 511 ––

```
typedef struct {
    SIGNAL_VARS
    SDL_Integer  Param1;
    SDL_Boolean  Param2;
} yPDef_sig;
typedef yPDef_sig  *yPDP_sig;
```

These types would represent a signal sig(Integer, Boolean).

As all signals starts with the components defined in `SIGNAL_VARS` it is possible to type cast a pointer to a signal, to the `xSignalNode` type, if only the components in `SIGNAL_VARS` is to be accessed.

Allocation of Data Areas for Signals

In `sctos.c` there are two functions, `xGetSignal` and `xReleaseSignal`, where data areas for signal are handled:

```
xSignalNode xGetSignal(
  xSignalIdNode  SType,
  SDL_PId        Receiver,
  SDL_PId        Sender )

void xReleaseSignal( xSignalNode  *S )
```

`xGetSignal` takes a reference to the `SignalIdNode` identifying the signal type and two PId values (sending and receiving process instance) and returns a signal instance. `xGetSignal` first looks in the avail list for the signal type (the component `AvailSignalList` in the `SignalIdNode` for the signal type) and reuses any available signal there. Only if the avail list is empty new memory is allocated. The component `VarSize` in the `SignalIdNode` for the signal type provides the size information needed to correctly allocate the `yPDef_`*`SignalName`* even though the type is unknown for the `xGetSignal` function.

The function `xReleaseSignal` takes the address of an `xSignalNode` pointer and returns the referenced signal to the avail list for the signal type. The `xSignalNode` pointer is then set to `0`.

The function `xGetSignal` is used:

- In generated code (output, set, reset)
- In a number of places in the library:
  SDL_Create
  SDL_SimpleReset
  SDL_Nextstate (to handle continuous signals)
- In the postmaster communication section and in the monitor to obtain signal instances.

The function `xReleaseSignal` is used by:

- SDL_Nextstate
- SDL_Stop, in both cases to release the signal that initiated the transition.

Overview of Output and Input of Signals

In this subsection the signal handling operation is only outlined. More details will be given in the section treating processes. See "Output and Input of Signals" on page 3088.

Signal instances are sent using the function SDL_Output. That function takes a signal instance and inserts it into the input port of the receiving process instance.

If the receiver is not already in the ready queue (the queue containing the processes that can perform a transition, but which have not yet been scheduled to do so) and the current signal may cause an immediate transition, the process instance is inserted into the ready queue.

If the receiver is already in the ready queue or in a state where the current signal should be saved, the signal instance is just inserted into the input port.

If the signal instance can neither cause a transition nor should be saved, it is immediately discarded (the data area for the signal instance is returned to the avail list).

The input port is scanned during nextstate operations, according the rules of SDL, to find the next signal in the input port that can cause a transition. Signal instances may then be saved or discarded.

There is no specific input function, instead this behavior is distributed both in the runtime library and in the generated code. The signal instance that should cause the next transition to be executed is removed from the input port in the main loop (the scheduler), immediately before the PAD function for the current process is called. The PAD function is the function where the behavior of the process is implemented and is part of the generated code. The assignment of the signal parameters to local SDL variables is one of the first actions performed by the PAD function.

The signal instance that caused a transition is released and returned to the avail list in the nextstate or stop action that ends the current transition.

Timers and Operations on Timers

A timer with parameters is represented by a type definition, where the timer parameters are defined, in exactly the same way as for a signal definition, see "Data Structure Representing Signals and Timers" on page 3070. At runtime, all timers that are set and where the timer time has not expired, are represented by a xTimerRec struct and a signal instance:

```
#define TIMER_VARS \
   xSignalNode   Pre; \
   xSignalNode   Suc; \
   int           Prio; \
   SDL_PId       Receiver; \
   SDL_PId       Sender; \
   xSignalIdNode NameNode; \
   GLOBALINSTID \
   ENVCHANNEL \
   SDL_Time      TimerTime;

typedef xTimerRec  *xTimerNode;

typedef struct xTimerStruct {
   TIMER_VARS
}  xTimerRec;
```

The TIMER_VARS is and must be identical to the SIGNAL_VARS macro, except for the TimerTime component last in the macro. A timer with parameters have yPDef_timername and yPDP_timername types in generated code exactly as a signal (see previous section), except that SIGNAL_VARS is replaced by TIMER_VARS.

During its life-time a timer have two different appearances. First it is a timer waiting for the timer time to expire. In that phase the timer is inserted in the `xTimerQueue`. When the timer time expires the timer becomes a signal and is inserted in the input port of the receiver just like any other signal. Due to the identical typedefs for `xSignalRec` and `xTimerRec`, there are no problems with type casting between `xTimerNode` and `xSignalNode` types.

When a timer is treated as a signal the components in the `xTimerRec` are used in the same ways as for a `xSignalRec`. While the timer is in the timer queue, the components are used as follows:

- **`Pre`** and **`Suc`** are pointers used to link the `xTimerRec` into the timer queue (the queue of active timers, see below).
- **`TimerTime`** is the time given in the `Set` operation.

The queue mentioned above, the timer queue for active timers is represented by the component xTimerQueue in the variable `xSysD`:

```
xTimerNode xTimerQueue;
```

The variable is initialized in the function `xInitKernel` in `sctsdl.c`. `xTimerQueue` is initialized it refers to the queue head of the timer queue.

The queue head is an extra element in the timer queue that does not represent a timer, but is introduced as it simplifies the algorithms for queue handling. The `TimerTime` component in the queue head is set to a very large time value (`xSysD.xMaxTime`).

The timer queue is thus a doubly linked list with a list head and it is sorted according to the timer times, so that the timer with lowest time is at the first position.

To optimize the case where there are many active timers, one extra pointer is introduced for each timer in the data area for the process instances. This pointer is used to refer to the active timer and to the timer signal up to the point where the signal is received in an input. A reset or active operation is then trivial, as there is a pointer to the timer object. If the pointer is NULL no timer is set.

For timers with parameters this model is extended, so a list of timers can be accessed from the process.

The xTimerRec structs are allocated and reused in the same way as signal.

From the SDL point of view, timers are handled in:

- Timer definitions
- Set and reset operations
- Timer outputs.

The timer output is the event when the timer time has expired and the timer signal is sent. After that, a timer signal is treated as an ordinary signal. These operations are implemented as follows:

```
void SDL_Set(
  SDL_Time     T,
  xSignalNode  S )
```

This function, which represents the Set operation, takes the timer time and a signal instance as parameters. It first uses the signal instance to make an implicit reset (see reset operation below) It then updates the TimerTime component in S and inserts S into the timer queue at the correct position.

The SDL_Set operation is used in generated code, together with xGetSignal, in much the same way as SDL_Output. First a signal instance is created (by xGetSignal), then timer parameters are assigned their values, and finally the Set operation is performed (by SDL_Set).

```
void SDL_Reset( xSignalNode  *TimerS )

void SDL_SimpleReset(
  xPrsNode       P,
  xSignalIdNode  TimerId )
```

Two functions are used to represent the SDL action reset. SDL_SimpleReset is used for timers without parameters and SDL_Reset for timers with parameters.

SDL_Reset uses the two functions xRemoveTimer and xRemoveTimerSignal to remove a timer in the timer queue and to remove a signal instance in the input port of the process. It then releases the signal instance given as parameter. This signal is only used to carry the parameter values given in the reset action.

The function SDL_SimpleReset is implemented in the same way as SDL_Reset, except that it creates its own signal instance (without parameters).

At a reset action the possibly found timer is removed from the timer queue and returned to the avail list. A found signal instance (in the input port) is removed from the input port and returned to the avail list for the current signal type.

```
static void SDL_OutputTimerSignal( xTimerNode  T )
```

The SDL_OutputTimerSignal is called from the main loop (the scheduler) when the timer time has expired for the timer first in the timer queue. The corresponding signal instance is then sent.

SDL_OutputTimerSignal takes a pointer to an xTimerRec as parameter, removes it from the timer queue and sends as an ordinary output using the function SDL_Output.

It can be checked if timer is active by using a call to the function SDL_Active. This function is used in generated code to represent the SDL operator active.

```
SDL_Boolean SDL_Active (
  xSignalIdNode TimerId,
  xPrsNode      P )
```

Note:

Only timers without parameters can be tested. This is a restriction in the Cadvanced/Cbasic SDL to C Compiler.

There is one more place where timers are handled. When a process instance performs a stop action all timers in the timer queue connected to this process instance are removed. This is performed by calling the function xRemoveTimer with the first parameter equal to 0.

Processes

Data Structure Representing Processes

A process instance is represented by two structs, an `xLocalPIdRec` and a struct containing both the general process data and the local variables and formal parameters of the process (`yVDef_`*`ProcessName`*), see also Figure 554. The reason for having both the `xLocalPIdRec` and the `yVDef_`*`ProcessName`* will be discussed under "Create and Stop Operations" on page 3085.

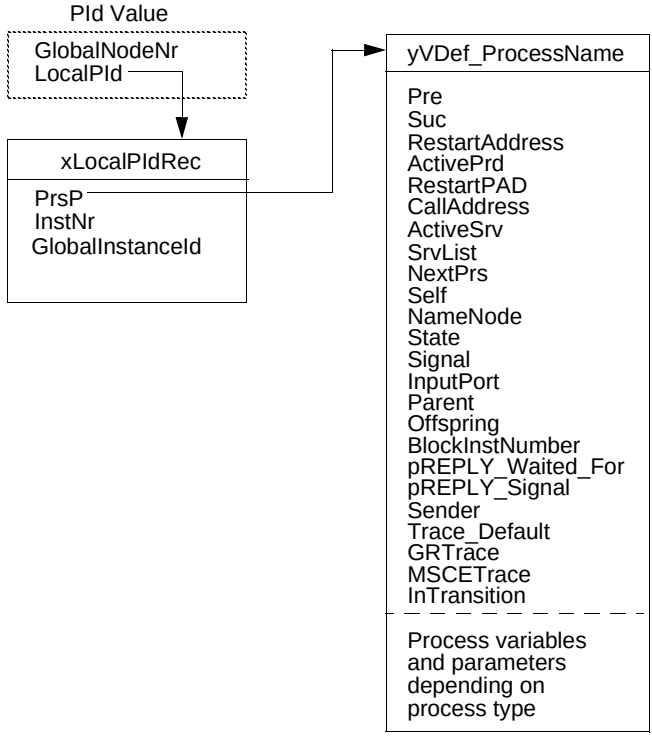


Figure 554: Representation of a process instance

The corresponding type definitions, which can be found in `scttypes.h`, are:

```
#ifdef XPRSSENDER
#define XPRSSENDERCOMP   SDL_PId  Sender;
#else
#define XPRSSENDERCOMP
#endif

#ifdef XTRACE
#define XTRACEDEFAULTCOMP   int Trace_Default;
#else
#define XTRACEDEFAULTCOMP
#endif

#ifdef XGRTRACE
#define XGRTRACECOMP   int GRTrace;
#else
#define XGRTRACECOMP
#endif

#ifdef XMSCE
#define XMSCETRACECOMP   int  MSCETrace;
#else
#define XMSCETRACECOMP
#endif

#if defined(XMONITOR) || defined(XTRACE)
#define XINTRANSCOMP   xbool InTransition;
#else
#define XINTRANSCOMP
#endif

#ifdef XMONITOR
#define XCALL_ADDR   int  CallAddress;
#else
#define XCALL_ADDR
#endif

#ifndef XNOUSEOFSERVICE
#define XSERVICE_COMP \
        xSrvNode ActiveSrv; xSrvNode SrvList;
#else
#define XSERVICE_COMP
#endif



#define PROCESS_VARS \
   xPrsNode       Pre; \
   xPrsNode       Suc; \
   int            RestartAddress; \
   xPrdNode       ActivePrd; \
   void (*RestartPAD) (xPrsNode  VarP); \
   XCALL_ADDR \
```

```
   XSERVICE_COMP \
   xPrsNode       NextPrs; \
   SDL_PId        Self; \
   xPrsIdNode     NameNode; \
   int            State; \
   xSignalNode    Signal; \
   xInputPortRec  InputPort; \
   SDL_PId        Parent; \
   SDL_PId        Offspring; \
   int            BlockInstNumber; \
   XSIGTYPE       pREPLY_Waited_For; \
   xSignalNode    pREPLY_Signal; \
   XPRSSENDERCOMP \
   XTRACEDEFAULTCOMP \
   XGRTRACECOMP \
   XMSCETRACECOMP \
   XINTRANSCOMP

typedef struct {
   xPrsNode      PrsP;
   int           InstNr;
   int           GlobalInstanceId;
}  xLocalPIdRec;

typedef xLocalPIdRec  *xLocalPIdNode;

typedef struct {
   int            GlobalNodeNr;
   xLocalPIdNode  LocalPId;
}  SDL_PId;

typedef struct xPrsStruct  *xPrsNode;

typedef struct xPrsStruct {
   PROCESS_VARS
} xPrsRec;
```

A `PId` value is thus a struct containing two components:

- The global node number
- A pointer to a `xLocalPIdRec` struct.

The use of the global node number is discussed in the chapter 57, *Building an Application*.

A **`xLocalPIdRec`** contains the following three components:

- **`PrsP`** of type `xPrsNode`. This component is a pointer to the `xPrsRec` struct that is part of the representation of the process instance.
- **`InstNr`** of type `int`. This is the instance number of the current process instance, which is used in the communication with the user in the monitor and in dynamic error messages.
- **`GlobalInstanceId`** is used in MSCE traces to have a unique identification of the process instance.

A **`xPrsRec`** struct contains the following components described below. As each `yVDef_ProcessName` struct contains the `PROCESS_VARS` macro as first item, it is possible to cast pointer values between a pointer to `xPrsRec` and a pointer to a `yVDef_ProcessName` struct.

- **`Pre`** and **`Suc`** of type `xPrsNode`. These components are used to link the process instance in the ready queue (see below).
- **`RestartAddress`** of type `int`. This component is used to find the appropriate SDL symbol to continue execute from.
- **`ActivePrd`** of type `xPrdNode`. This is a pointer to the `xPrdRec` that represents the currently executing procedure called from this process instance. The pointer is 0 if no procedure is currently called.
- **`RestartPAD`**, which is a pointer to a `PAD` function. This component refers to the `PAD` function where to execute the sequence of SDL symbols. `RestartPAD` is used to handle inheritance between process types.
- **`CallAddress`** of type `int`. This component contains the symbol number of the procedure call currently executed by this process.
- **`ActiveSrv`** of type `xSrvNode`. This component contains a reference to the currently active service (or latest active service) in this process.

- **SrvList** of type xSrvNode. This component contains a reference to the first service contained in this process. The component NextSrv in the struct representing a service can be used to find next active service in the process.
- **NextPrs** of type xPrsNode. This component is used to link the process instance either in the active list or in the avail list for this process type. The start of these two lists are the components ActivePrsList and AvailPrsList in the IdNode representing the current process type.
- **Self** of type SDL_PId. This is the PId value of the current process instance.
- **NameNode** of type xPrsIdNode. This is a pointer to the PrsIdNode representing the current process or process instantiation.
- **State** of type int. This component contains the int value used to representing the current state of the process instance.
- **Signal** of type xSignalNode. This is a pointer to a signal instance. The referenced signal is the signal that will cause the next transition by the current process instance, or that caused the transition that is currently executed by the process instance.
- **InputPort** of type xInputPortRec. This is the queue head in the doubly linked list that represents the input port of the process instance. The signals are linked in this list using the Pre and Suc components in the xSignalRec struct.
- **Parent** of type SDL_PId. This is the PId value of the parent process (according to the rules of SDL). A static process instance has parent equal to NULL.
- **Offspring** of type SDL_PId. This is the PId value of the latest created process instance (according to the rules of SDL). A process instance that has not created any processes has offspring equal to NULL.
- **BlockInstNumber** of type int. If the process is part of a block instance set, this component indicates which of the blocks that the process belongs to.

- **pREPLY_Waited_For** of type xSignalIdNode. When a process is waiting in the implicit state for the pREPLY signal in a RPC call, this components is used to store the IdNode for the expected pREPLY signal.
- **pREPLY_Signal** of type xSignalNode. When a process receives a pCALL signal, i.e. accepts a RPC, it immediately creates the return signal, the pREPLY signal. This component is used to refer to this pREPLY signal until it is sent.
- **Sender** of type SDL_PId. This component represents the SDL concept Sender.
- **Trace_Default** of type int. This component contains the current value of the trace defined for the process instance.
- **GRTrace** of type int. This component contains the current value of the GR trace defined for the process instance.
- **MSCETrace** of type int. This component contains the current MSCE trace value for the process instance.
- **InTransition** of type xbool. This component is true while the process is executing a transition and it is false while the process is waiting in a state. The monitor system needs this information to be able to print out relevant information.

The Ready Queue, Scheduling

The ready queue is a doubly linked list with a head. It contains the process instances that can execute an immediate transition, but which has not been allowed to complete that transition. Process instances are inserted into the ready queue during output operations and nextstate operations and are removed from the ready queue when they execute the nextstate or stop operation that ends the current transition. The head in the ready queue, which is an object in the queue that does not represent any process but is inserted only to simplify the queue operations, is referenced by the xSysD component:

```
xPrsNode   xReadyQueue;
```

This component is initiated in the function xInitKernel and used throughout the runtime library to reference the ready queue.

Scheduling of events is performed by the function `xMainLoop`, which is called from the `main` function after the initialization is performed.

```
void xMainLoop()
```

The strategy to have all interesting queues (the ready queue, the timer queue, and the input ports) sorted in the correct order is used in the library. Sorting is thus performed when an object is inserted into a queue, which means that scheduling is a simple task: select the first object in the timer queue or in the ready queue and submit it for execution.

There are several versions of the body of the endless loop in the function `xMainLoop`, which are used for different combinations of compilation switches. When it comes to scheduling of transitions and timer outputs they all have the following outline:

```
while (1) {
  if ( xTimerQueue->Suc->TimerTime <= SDL_Now() )
    SDL_OutputTimerSignal( xTimerQueue->Suc );
  else if ( xReadyQueue->Suc != xReadyQueue) {
    xRemoveFromInputPort(xReadyQueue->Suc->Signal);
    xReadyQueue->Suc->Sender =
       xReadyQueue->Suc->Signal->Sender;
   (*xReadyQueue->Suc->RestartPAD)(xReadyQueue->Suc);
  }
}
```

or, in descriptive terms:

```
while (1) {
  if ( there is a timer that has expired )
    send the corresponding timer signal;
  else if ( there is a process that can execute
            a transition ) {
    remove the signal causing the transition
      from input port;
    set up Sender in the process to Sender of
      the signal;
    execute the PAD function for the process;
  }
}
```

The different versions of the main loop handle different combinations of compilation switches. Other actions necessary in the main loop are dependent of the compilation switches. Example of such actions are:

- Handling of the monitor
- Calling the `xInEnv` function

- Handling real time or simulated time
- Delay execution up to the next scheduled event
- Handling enabling conditions and continuous signals that need to be recalculated.

Create and Stop Operations

A process instance is, while it is active, represented by the two structs:

- `xLocalPIdRec`
- The `yVDef_ProcessName` struct.

These two structs are dynamically allocated. A `PId` value is also a struct (not allocated) containing two components, `GlobalNodeNr` and `LocalPId`, where `LocalPId` is a pointer to the `xLocalPIdRec`. Figure 555 shows how the `xLocalPIdRec` and the `yVDef_ProcessName` structs representing a process instance are connected.

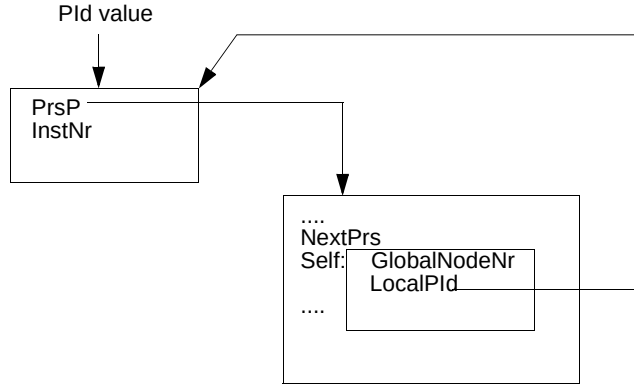


Figure 555: A xLocalPIdRec and a yVDef_ProcessName representing a Process instance

When a process instance performs a stop action, the memory used for the process instance should be reclaimed and it should be possible to reuse in subsequent create actions. After the stop action, old (invalid) `PId` values might however be stored in variables in other process instances.

If a signal is sent to such an old `PId` value, that is, to a stopped process instance, it should be possible to find and perform appropriate actions.

If the complete representation of a process instance is reused then this will not be possible. There must therefore remain some little piece of information and thus some memory for each process instance that has ever existed. This is the purpose of the `xLocalPIdRec`. These structs will never be reused. Instead the following (see Figure 556) will happen when the process instance in Figure 555 performs a stop action.

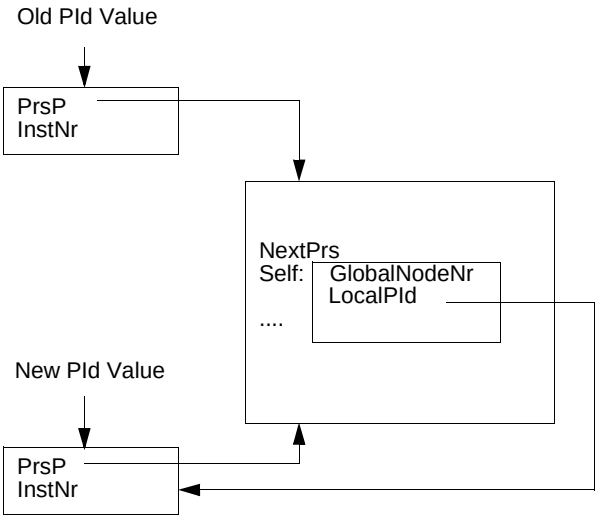


Figure 556: The memory structure after the process in Figure 555 has performed a stop action

A new `xLocalPIdRec` is allocated and its `PrsP` references the `yVDef_ProcessName` (`InstNr` is 0). The `Self` component in the `yVDef_ProcessName` is changed to reference this new `xLocalPIdRec`. The old `xLocalPIdRec` still references the `yVDef_ProcessName`. The `yVDef_ProcessName` is entered into the avail list for this process type.

To reuse the data area for a process instance at a create operation it is only necessary to remove the `yVDef_ProcessName` from the avail list and update the `InstNr` component in the `xLocalPIdRec` referenced by `Self`.

Using this somewhat complicated structure to represent process instances allows a simple test to see if a `PId` value refers to an active or a stopped instance:

If P is a PId variable then the following expression:

```
P.LocalPId == P.LocalPId->PrsP->Self.LocalPId
```

is true if the process instance is active and false if it is stopped.

The basic behavior of the create and stop operations is performed by the functions SDL_Create and SDL_Stop.

```
void SDL_Create(
  xSignalNode  StartUpSig,
  xPrsIdNode   PrsId )

void SDL_Stop( xPrsNode  PrsP )
```

To create a process instance takes three steps performed in generated code:

1. Call xGetSignal to obtain the start-up signal.
2. Assign the actual process parameters to the start up signal parameters.
3. Call SDL_Create with the start-up signal as parameter, together with the PrsIdNode representing the process to be created.

In xGetProcess the process instance is removed from the avail list of the process instance set (the component AvailPrsList in the PrsIdNode representing the process instance set), or if the avail list is empty new memory is allocated.

The process instance is linked into the list of active process instances (the component ActivePrsList in the PrsIdNode representing the process instance set). Both the avail list and the active list are single linked lists (without a head) using the component NextPrs in the yVDef_ProcessName struct as link.

To have an equal treatment of the initial transition and other transitions, the start state is implemented as an ordinary state with the name “start state” It is represented by 0. To execute the initial transition a “startup” signal is sent to the process. The start state can thus be seen as a state with one input of the startup signal and with save for all other signals. This implementation is completely transparent in the monitor, where startup signals are never shown in any way.

Note:

The actual values for FPARs are passed in the startup signal.

Two `IdNodes` that are not part of the symbol table tree are created to represent a start state and a startup signal.

```
xStateIdNode   xStartStateId;
xSignalIdNode  xStartUpSignalId;
```

These `xSysD` components are initialized in the function `xInitSymbolTable`, which is part of `sctsdl.c`.

At a stop operation the function `SDL_Stop` is called. This function will release the signal that caused the current transition and all other signals in the input port. It will also remove all timers in the timer queue that are connected to this process instance by calling `xRemoveTimer` with the first parameter equal to `0`. It then removes the process executing the stop operation from the ready queue and from the active list of the process type and returns the memory to the avail list of the current process instance set.

Output and Input of Signals

There are three actions performed in generated code to send a signal. First `xGetSignal` is called to obtain a data area that represents the signal instance, then the signal parameters are assigned their values and finally the function `SDL_Output` is called to actually send the signal. First in the `SDL_Output` function there are a number of dynamic tests (check if receiver in TO-clause is not `NULL` and not stopped, check if there is a path to the receiver). If the output does not contain any TO-clause and the Cadvanced/Cbasic SDL to C Compiler has not been able to calculate the receiver, the `xFindReceiver` function is called to calculate the receiver according to the rules of SDL.

Next, in `SDL_Output` signals to the environment are handled. Three cases can be identified here:

1. The environment function `xOutEnv` is called.
2. The corresponding function that sends signals via the SDL Suite communication mechanism (`xOutPM`) is called.
3. The signal is inserted into the input port of the process representing the environment (`xEnv`).

Finally, internal signals in the SDL system are treated. Here also three cases can be identified (how this is evaluated is described last in this subsection):

1. The signal can cause an immediate transition by the receiver.
2. The signal should be saved.
3. The signal should be immediately discarded.

If the signal can cause an immediate transition, the signal is inserted into the input port of the receiver, and the receiving process instance is inserted into the ready queue.

If the signal should be saved, the signal is just inserted into the input port of the receiver.

If the signal should be discarded, the function `xReleaseSignal` is called to reused the data area for the signal.

When a signal is identified to be the signal that should cause the next transition by the current process instance (at an Output or Nextstate operation), the component `Signal` in the `yVDef_ProcessName` for the process is set to refer to the signal. The signal is still part of the input port list.

When the transition is to be executed, the signal is removed from the input port in the main loop (see "The Ready Queue, Scheduling" on page 3083) immediately before the `PAD` function for the process is called.

First in the `PAD` function, the parameters of the signal are copied to the local variables according to the input statement. In the ending Nextstate or Stop operation of the transition the signal instance is returned to the avail list.

Evaluating How To Handle a Received Signal

There are two places in the run-time kernel where it is necessary to evaluate how to handle signals (input, save, discard,...):

- At an Output operation to a currently idle process.
- At a Nextstate operation, when the process have signals in the input port.

This calculation is implemented in the run-time kernel function `xFindInputAction`.

```
typedef unsigned char xInputAction;
#define xDiscard         (xInputAction)0
#define xInput           (xInputAction)1
#define xSave            (xInputAction)2
#define xEnablCond       (xInputAction)3
#define xPrioInput       (xInputAction)4

static xInputAction xFindInputAction(
  xSignalNode  SignalId,
  xPrsNode     VarP,
  xbool        CheckPrioInput )
```

The parameters of this function is:

- `SignalId`, which is a pointer to a signal.
- `VarP`, which is a pointer to a process instance.
- `CheckPrioInput`, which is a boolean value indicating is the function should check only for priority inputs or for ordinary inputs.

As a result the function should return:

- The action that should be performed for this signal (input, save,...), taking all information about this process into account, like inheritance between processes, virtual - redefined transitions and so on.
- If the function result is `xInput` or `xPrioInput`, then the `RestartPAD` and `RestartAddr` components in the `VarP` struct should be updated with information about where this input can be found.

After this last update the correct transition can be started by the scheduler by just calling the function referenced by `RestartPAD`, which the as first action performs switch `RestartAddr` and starts execute the input symbol.

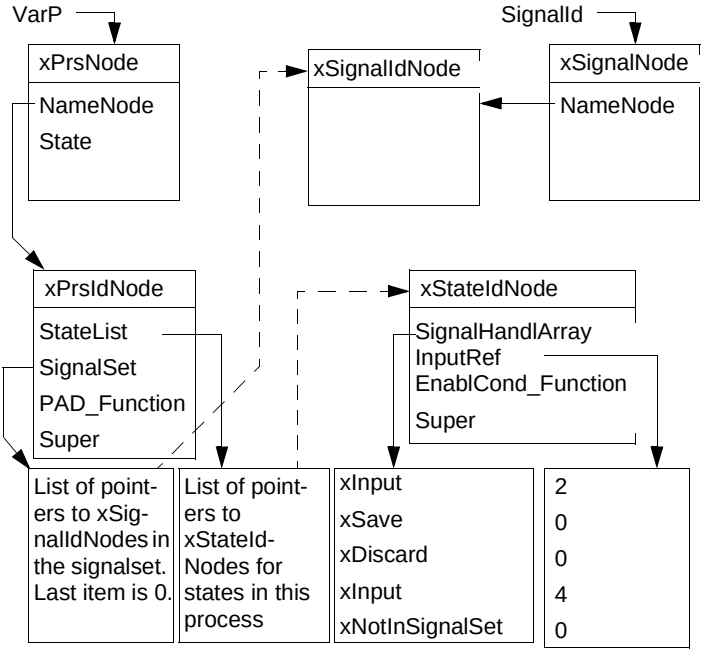


Figure 557: Data structure used to evaluate the xFindInputAction

The algorithm to find the `InputAction`, the `RestartAddr`, and the `RestartPAD` is as follows:

1. Let `ProcessId` become `yVarP->NameNode` and let `StateId` become `ProcessId->StateList[yVarP->State]`.
2. In `ProcessId->SignalSet` find the index (`Index`) where `SignalId->NameNode` is found. If the signal is not found, this signal is not in the signal set of the process, and the algorithm terminates returning the result `xDiscard`.
3. `StateId->SignalHandlArray[Index]` now gives the action to be performed. If this value is `xEnablCond`, then the function `StateId->EnablCond_Function` is called. This function returns either `xInput` or `xSave`.

4. If the result from step 3 is `xInput`, the algorithm terminates returning this value. `yVarP->RestartAddr` is also updated to `StateId->InputRef[Index]`, while `yVarP->RestartPAD` is updated to `ProcessId->PAD_Function`.

   If the result from step 3 is `xSave`, the algorithm terminates returning this value.

   If the result from step 3 is `xDiscard` and `ProcessId->Super` equal to `NULL`, then the algorithm terminates returning this value.

   If the result from step 3 is `xDiscard` and `ProcessId->Super` not equal to `NULL`, then we are in a process type that inherits from another process type. We then have to perform step 2 - 4 again, with `ProcessId` assigned the value `ProcessId->Super` and `StateId` assigned the value `StateId->Super`.

Nextstate Operations

The nextstate operation is implemented by the `SDL_Nextstate` function, where the following actions are performed:

1. The signal that caused the current transition (component `Signal` in the `yVDef_ProcessName`) is released and the state variable (component `State` in the `yVDef_ProcessName`) is updated to the new state.

2. Then the input port of the process is scanned for a signal that can cause a transition. During the scan signals might be saved or discarded until a signal specified in an input is found. Priority inputs are treated according to the rules of SDL.

3. If no signal that can cause a transition is found, a check is made if any continuous signal can cause a transition (see "Enabling Conditions and Continuous Signals" on page 3093). The process is thereafter removed from the ready queue.

4. If any signal (or continuous signal) can cause a transition then the process is re-inserted into the ready queue again at a position determined by its priority, else if the new state contains any continuous signal or enabling condition with an expression that might change its value during the time the process is in the state (view, import...), the process is inserted into the check list (see also "Enabling Conditions and Continuous Signals" on page 3093).

Decision and Task Operations

Decision and Task operations are implemented in generated code, except for the Trace-functions implemented in the `sctutil.c` and `sctmon.c` files and for informal and any decisions that uses some support functions in `sctmon.c`. A Decision is implemented as a C if-statement, while the assignments in a Task are implemented as assignments or function calls in C.

Compound Statements

A compound statement without variable declarations is translated just to the sequence of action it contains, while a compound statement with variable declarations is translated in the same way as an SDL procedure (without parameters). Statements within a compound statement are translated according to the normal rules. The new statement types in compound statements are translated as:

- **if** in SDL is translated to **if** in C
- **decision** in compound statements is translated as ordinary decisions.
- **for** loops, **continue**, and **break** are all translated using **goto** in C.

Enabling Conditions and Continuous Signals

The expressions involved in continuous signals and enabling conditions are implemented in generated code in functions called `yCont_`*`StateName`* and `yEnab_`*`StateName`*. These functions are generated for each state containing continuous signals respectively enabling conditions. The functions are referenced through the components `ContSig_Function` and `EnablCond_Function` in the `StateIdNode` for the state. These components are `0` if no corresponding functions are generated.

The `EnablCond_Functions` are called from the function `xFindInputAction`, which is called from `SDL_Output` and `SDL_Nextstate`. If the enabling condition expression for the current signal is true then `xInput` is returned else `xSave` is returned. This information is then used to determine how to handle the signal in this state.

The `ContSig_Functions` are called from `SDL_Nextstate`, if the component `ContSig_Function` is not `0` and no signal that can cause an immediate transition is found during the input port scan. A `ContSig_Function` has the following prototype:

```
void ContSig_Function_Name (
  void *, int *, xIdNode *, int *);
```

where the first parameter is the pointer to the `yVDef_ProcessName`. The remaining parameters are all out parameters; the second contains the priority of the continuous signal with highest priority (=lowest value) that has an expression with the value true. Otherwise <0 is returned here. The third and fourth is only defined the second parameter >=0; the third is the `IdNode` for the process/procedure where the actual continuous signal can be found and the fourth is the `RestartAddress` connected to this continuous signal.

If a continuous signal expression with value true is found, a signal instance representing the continuous signal is created and inserted in the input port, and is thereafter treated as an ordinary signal. The signal type is continuous signal and is represented by an `SignalIdNode` (referenced by the variable `xContSigId`).

The check list is a list that contains the processes that wait in a state where enabling conditions or continuous signals need to be repeatedly recalculated.

A process is inserted into the check list if:

1. It enters a state containing enabling conditions and/or continuous signals and
2. No signal or continuous signal can cause an immediate transition and
3. One or several of the expressions in the enabling conditions or continuous signals can change its value while the process is in the state (view, import, now, ...)

The component `StateProperties` in the `StateIdNode` reflects if any such expression is present in the state.

The check list is represented by the `xSysD` component:

```
xPrsNode  xCheckList;
```

The behavior of enabling conditions and continuous signals is in SDL modeled by letting the process repeatedly send signals to itself, thereby to repeatedly entering the current state. In the implementation chosen here, nextstate operations are performed “behind the scene” for all pro-

cesses in the check list directly after a call to a PAD function is completed, that is directly after a transition is ended and directly after a timer output. This is performed by calling the function xCheckCheckList in the main loop of the program.

View and Reveal

A view expression is part of an expression in generated code and implemented by calling the function SDL_View.

```
void * SDL_View (
  xViewListRec *VList,
  SDL_PId       P,
  xbool         IsDefP,
  xPrsNode      ViewingPrs,
  char *        Reveal_Var
  int           SortSize);
```

- **VList** is a list of all revealed variables in this block.
- **P** is the PId expression given in the view statement.
- **IsDefP** is 1 is the view expression contained a PId value, 0 otherwise.
- **ViewingPrs** is the process instance performing the view operation.
- **Reveal_Var** is the name of the revealed variable as a string. The Reveal_Var parameter is only used in error messages and is remove under certain conditions.
- **SortSize** is the size of the data type of the viewed variable.

The SDL_View function performs a test that the view expression is not NULL, refers to a process in the environment, or to a stopped process instance. If no errors are found the address of the revealed variable is returned as result from the SDL_View function. Otherwise the address of a variable containing only zeros is returned.

Import, Export, and Remote Variables

For an exported variable there are two components in the yVDef_ProcessName struct. One for the current value of the variable and one for the currently exported value of the variable. For each exported variable there will also be a struct that can be linked into a list in

the corresponding `RemoteVarIdNode`. This list is then used to find a suitable exporter of a variable in an import action.

An export action is a simple operation. The current value of the variable is copied to the component representing the exported value. This is performed in generated code.

An import action is more complicated. It involves mainly a call of the function `xGetExportAddr`:

```
void * xGetExportAddr (
  xRemoteVarIdNode RemoteVarNode,
  SDL_PId          P,
  xbool            IsDefP,
  xPrsNode         Importer )
```

`RemoteVarNode` is a reference to the `RemoteVarIdNode` representing the remote variable (implicit or explicit), `P` is the PId expression given in the import action and `IsDef` is 0 or 1 depending on if any `PId` expression was given in the import action or not, Importer is the importing process instance. The `xGetExportAddr` will check the legality of the import action and will, if no `PId` expression is given, calculate which process it should be imported from.

If no errors are found the function will return the address where the exported value can be found. This address is then casted to the correct type (in generated code) and the value is obtained. If no process possible to import from is found, the address of a variable containing only zeros is returned by the `xGetExportAddr` function.

Note:

The strategy for import actions is in one sense not equal to the model for import given in the SDL recommendation. An import action is in the recommendation modeled as a signal sent from the importing process to the exporting process asking for the exported value, and a signal with this value sent back again. The synchronization effects by this signal communication is lost in the implementation model we have chosen. Instead our model is much easier and faster and the primary part of the import action, to obtain the exported value, is the same.

Services

Data Structure Representing Services

A service is represented by a struct type. The `xSrvRec` struct defined in `scttypes.h`, is, just like `xPrsRec` for processes, a struct containing general information about a service, while the parameters and variables of the service are defined in generated code in the same way as for processes.

In `scttypes.h` the following types concerning procedures can be found:

```
#ifdef XMONITOR
#define XCALL_ADDR   int  CallAddress;
#else
#define XCALL_ADDR
#endif

#define SERVICE_VARS \
   xSrvNode       NextSrv; \
   xPrsNode       ContainerPrs; \
   int            RestartAddress; \
   xPrdNode       ActivePrd; \
   void (*RestartPAD) (xPrsNode  VarP); \
   XCALL_ADDR \
   xSrvIdNode     NameNode; \
   int            State; \
   XSIGTYPE       pREPLY_Waited_For; \
   xSignalNode    pREPLY_Signal; \
   XINTRANSCOMP
#ifndef XNOUSEOFSERVICE
typedef struct xSrvStruct *xSrvNode;
#endif

#ifndef XNOUSEOFSERVICE
typedef struct xSrvStruct {
   SERVICE_VARS
}  xSrvRec;
#endif
```

In generated code yVDef_ProcedureName structs are defined according to the following:

```
typedef struct {
   SERVICE_VARS
   components for FPAR and DCL
}  yVDef_ServiceName;
```

The components in the `xSrvRec` are used as follows:

- **`NextSrv`** of type `xSrvNode`. Reference to next service contained in this process.
- **`ContainerPrs`** of type `xPrsNode`. Reference to the process instance containing this service.
- **`RestartAddress`** of type `int`. This component is used to find the appropriate SDL symbol to continue execution from.
- **`ActivePrd`** of type `xPrdNode`. This is a pointer to the xPrdRec that represents the currently executing procedure called from this service instance. The pointer is 0 if no procedure is currently called.
- **`RestartPAD`**, which is a pointer to a PAD function. This component refers to the PAD function where to execute the sequence of SDL symbols. RestartPAD is used to handle inheritance between service types.
- **`CallAddress`** of type `int`. This component contains the symbol number of the procedure call performed from this procedure (if any).
- **`NameNode`** of type `xSrvIdNode`. This is a pointer to the IdNode representing the service or service instantiation.
- **`State`** of type `int`. This component contains the int value used to represent the current state of the service instance.
- **`pREPLY_Waited_For`** of type `xSignalIdNode`. When a service is waiting in the implicit state for the pREPLY signal in a RPC call, this components is used to store the IdNode for the expected pREPLY signal.

- **pREPLY_Signal** of type xSignalNode. When a service receives a pCALL signal, i.e. accepts a RPC, it immediately creates the return signal, the pREPLY signal. This component is used to refer to this pREPLY signal until it is sent.
- **InTransition** of type xbool. This component is true while the service is executing a transition and it is false while the service is waiting in a state. The monitor system needs this information to be able to print out relevant information.

Executing Transitions in Services

From the scheduler's point view, it is not of interest if a process contains services or not. It is still the process instance that is scheduled in the ready queue and the PAD function of the process that is to be called to execute a transition. The PAD function for a process containing services performs three different actions:

- Assign default value to variables declared at the process level
- Create one service instance for each service or service instantiation in the process.
- Calls the proper PAD function for a service to execute transitions.

The structure for a PAD function for a process with services are as follows:

```
YPAD_FUNCTION(yPAD_z00_P1)
{
  YPAD_YSVARP
  YPAD_YVARP(yVDef_z00_P1)
  YPRSNAME_VAR("P1")
  LOOP_LABEL_SERVICEDECOMP
  CALL_SERVICE

/*-----
* Initialization (no START symbol)
------*/
  BEGIN_START_TRANSITION(yPDef_z00_P1)
  yAssF_SDL_Integer(yVarP->z002_Global,
     SDL_INTEGER_LIT(10), XASS);
  START_SERVICES
}
```

where LOOP_LABEL_SERVICEDECOMP and BEGIN_START_TARNSITION are empty macros, i.e. expanded to no code. The yAss_SDL_Integer statement in an assignment of a default value to a process variable.

The macro CALL_SERVICE is expanded to:

```
if (yVarP->ActiveSrv != (xSrvNode)0) {
  (*yVarP->ActiveSrv->RestartPAD)(VarP);
  return; \
}
```

that is to a call of the PAD function of service reference by ActiveSrv.

The macro START_SERVICE is expanded to a call to the function `xStart_Services`, which can be found in `sctsdl.c`.The function creates the service instances, sets up the ActiveSrv pointer for the process to the first service, and then schedules the process for a new transition. This means that the next action performed by the system will be the start transition by the first service instance. When the first service executes a nextstate or stop action in the end of its start transition, the process will be scheduled again to execute the start transition of the second service, and so on until all services in the process has executed its start transitions.

For ordinary transitions, i.e. reception of a signal, it is obvious from the code above that the ActiveSrv pointer is essential. It should refer to the service instance that is to be executed. When a signal is to be received by a process, it is the function xFindInputAction (in `sctsdl.c`) that determines how to handle the signal and if it is to be received, where is the code for that transition. This function now also determines and sets up the ActiveSrv pointer.

Procedures

Data Structure Representing Procedures

A procedure is represented by a struct type. The `xPrdRec` struct defined in `scttypes.h`, is, just like `xPrsRec` for processes, a struct containing general information about a procedure, while the parameters and variables of the procedure are defined in generated code in the same way as for processes.

In `scttypes.h` the following types concerning procedures can be found:

```
#define PROCEDURE_VARS \
   xPrdIdNode   NameNode; \
   xPrdNode     StaticFather; \
   xPrdNode     DynamicFather; \
   int          RestartAddress; \
   XCALL_ADDR \
   void (*RestartPAD) (xPrsNode  VarP); \
   xSignalNode  pREPLY_Signal; \
   int          State;

typedef struct xPrdStruct  *xPrdNode;

typedef struct xPrdStruct {
   PROCEDURE_VARS
}  xPrdRec;
```

In generated code yVDef_ProcedureName structs are defined according to the following:

```
typedef struct {
   PROCEDURE_VARS
   components for FPAR and DCL
}  yVDef_ProcedureName;
```

The components in the xPrdRec are used as follows:

- **NameNode** of type xPrdIdNode. This is a pointer to the IdNode representing the procedure type.
- **StaticFather** of type xPrdNode. This is a pointer that represents the scope hierarchy of procedures (and the process at the top), which is used when a procedure instance refers to non-local variables. An example is shown in Figure 558 on page 3103. StaticFather == 0 means that the static father is the process.
- **DynamicFather** of type xPrdNode. This is a pointer that represents that this procedure is called by the referenced procedure. DynamicFather == 0 means that this procedure was called from the process. This component is also used to link the xPrdRec in the avail list for the procedure type.
- **RestartAddress** of type int. This component is used to find the appropriate SDL symbol to continue execution from.
- **CallAddress** of type int. This component contains the symbol number of the procedure call performed from this procedure (if any).
- **RestartPRD** is a pointer to a procedure function. This component refers to the PRD function where to execute the next sequence of SDL symbols. RestartPRD is used to handle inheritance between procedures.
- **pREPLY_Signal** of type xSignalNode. When a process receives a pCALL signal, i.e. accepts a RPC, it immediately creates the return signal, the pREPLY signal. This component is used to refer to this pREPLY signal until it is sent.
- **State** of type int. This is the value representing the current state of the procedure instance.

In Figure 558 on page 3103 an example of the structure of yVDef_ProcedureName after four nested procedure calls are presented. Note that procedure Q is declared in the process, procedure R and S in Q and T in S.

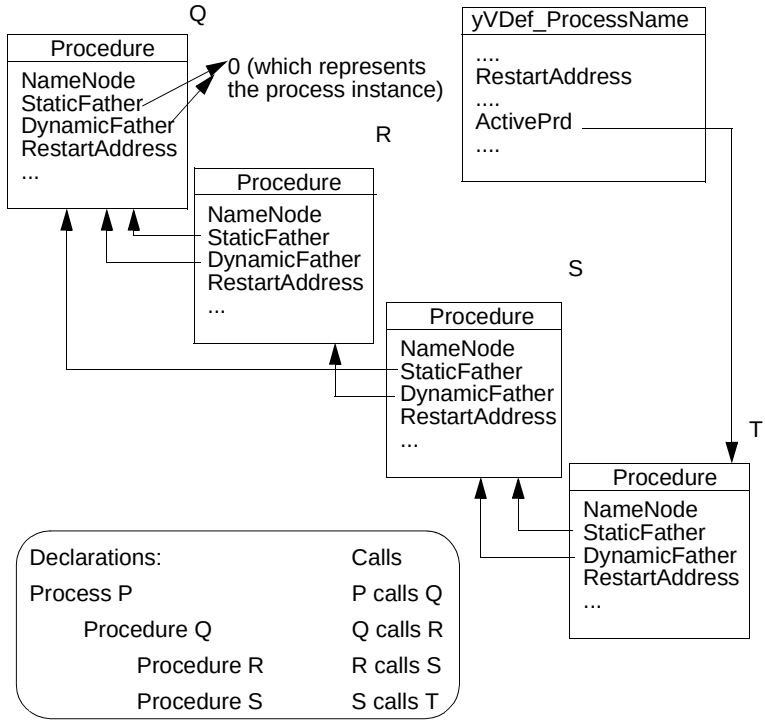


Figure 558: Structure of `yVDef_ProcedureName` *after four nested procedure calls*

The SDL procedures are partly implemented using C functions and partly using the structure shown above. Each SDL procedure is represented by a C function, which is called to execute actions defined in the procedure. This function corresponds to the `PAD` function for processes. The formal parameters and the variables are however implemented using a struct defined in generated code. The procedure stack for nested procedure calls is implemented using the components `StaticFather` and `DynamicFather`, and does not use the C function stack.

Calling and Returning from Procedures

Procedure calls and procedure returns are handled by three functions, one handling allocation of the data areas for procedures:

```
xPrdNode  xGetPrd( xPrdIdNode  PrdId )
```

and two functions called from generated code at a procedure call and a procedure return:

```
void xAddPrdCall(
  xPrdNode  R,
  xPrsNode  VarP,
  int       StaticFatherLevel,
  int       RestartAddress )

void xReleasePrd (xPrsNode  VarP)
```

A procedure call in SDL is in C represented by the following steps:

1. Calling `xGetPrd` to obtain a data area for the procedure.
2. Assigning procedure parameters to the data area.
3. Calling `xAddPrdCall` to link the procedure into the static and dynamic chains.
4. Calling the C function modeling the SDL procedure, i.e. the `yProcedureName` function.

The parameters to `xAddPrdCall` are as follows:

- **`R`**. A reference to the `xPrdNode` obtained from the call of `xGetPrd`.
- **`VarP`**. A reference to the yVDef_ProcessName, i.e. the data area for variables and parameters of the process (even if it is a procedure that performed the procedure call).
- **`StaticFatherLevel`**. This is the difference in declaration levels between the caller and the called procedure. This information is used to set up the `StaticFather` component correctly.
- **`RestartAddress`**. This is the symbol number of the SDL symbol directly after the procedure call. The symbol number is the switch case label generated for all symbols.

The `xGetPrd` returns a pointer to an `xPrdRec`, which can then be used to assign the parameter values directly to the components in the data

area representing the formal parameters and variables of the procedure. Note that IN/OUT parameters are represented as addresses in this struct.

A procedure return is in generated code represented by calling the `xReleasePrd` followed by return 0, whereby the function representing the behavior of the SDL procedure is left.

The function representing the behavior of the SDL procedure is returned in two main situations:

- When an SDL `Return` is reached (the function returns 0)
- When a `Nextstate` is reached (the function returns 1).

If 0 is returned then the execution should continue with the next SDL symbol after the procedure call, while if 1 is returned the execution of the process instance should be terminated and the scheduler (main loop) should take control. This could mean that a number of nested SDL procedure calls should be terminated.

To continue to execute at the correct symbol when a procedure should be resumed after a nextstate operation, the following code is introduced in the `PAD` function for processes containing procedure calls:

```
while ( yVarP->ActivePrd != (xPrdNode)0 )
  if ((*yVarP->ActivePrd->RestartPRD)(VarP))
    return;
```

This means that uncompleted procedures are resumed one after one from the bottom of the procedure stack, until all procedures are completed or until one of them returns 1, i.e. executes a nextstate operation, at which the process is left for the scheduler again.

Channels and Signal Routes

The `ChannelIdNodes` for channels, signal routes, and gates are used in the functions `xFindReceiver` and `xIsPath`, which are both called from `SDL_Output`, to find the receiving process when there is no TO clause in the Output statement, respectively to check that there is a path to the receiver in the case of a TO clause in the Output statement. In both cases the paths built up using the `ToId` components in the `IdNodes` for processes, channels and signal routes are followed. To show the structure of these paths we use the small SDL system given in Figure 559.

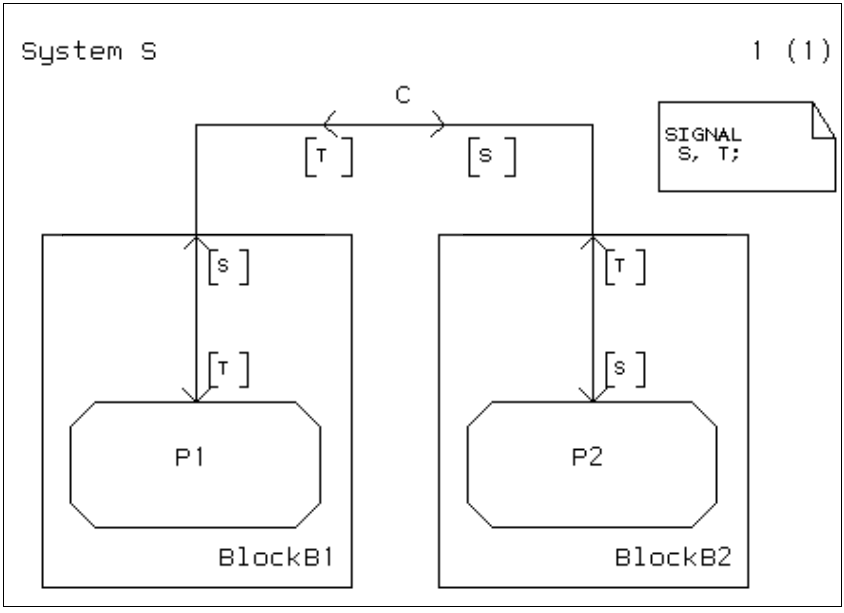


Figure 559: A small SDL system

During the initialization of the system, the symbol table is built up. The part of the symbol table starting with the system will then have the structure outlined in Figure 560. As we can see in this example the declarations in the SDL system are directly reflected by `IdNodes`.

Note:

Each channel and signal route is represented by two `IdNodes`, one for each direction. This is also true for an unidirectional channel or signal route. In this case the signal set will be empty for the unused direction.

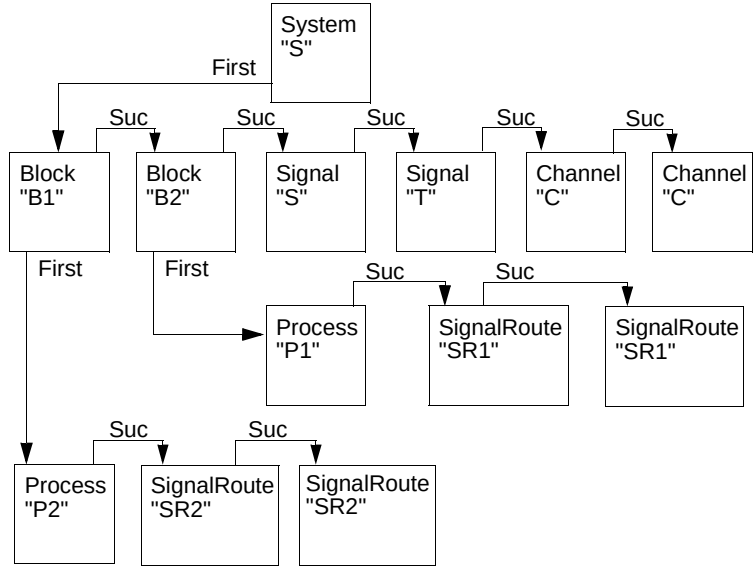


Figure 560: The symbol table tree for the system in Figure 559

Each IdNode representing a process, a signal route, or a channel will have a component `ToId`. A `ToId` component is an address to an array of references to `IdNodes`. The size of this array is dependent on the number of items this object is connected to. A process that has three outgoing signal routes will have a `ToId` array which can represent three pointers plus an ending `0` pointer.

In the example in Figure 559 and Figure 560 there is no branching, so all `ToId` arrays will be of the size necessary for two pointers. Figure 561 shows how the `IdNodes` for the processes, signal routes and channels are connected to form paths, using the components `ToId`. In this case only simple paths are found (one from P1, via SR1, C, SR2, to P2, and

one in the reverse direction). The generalization of this structure to handle branches is straightforward and discussed in the previous paragraph.

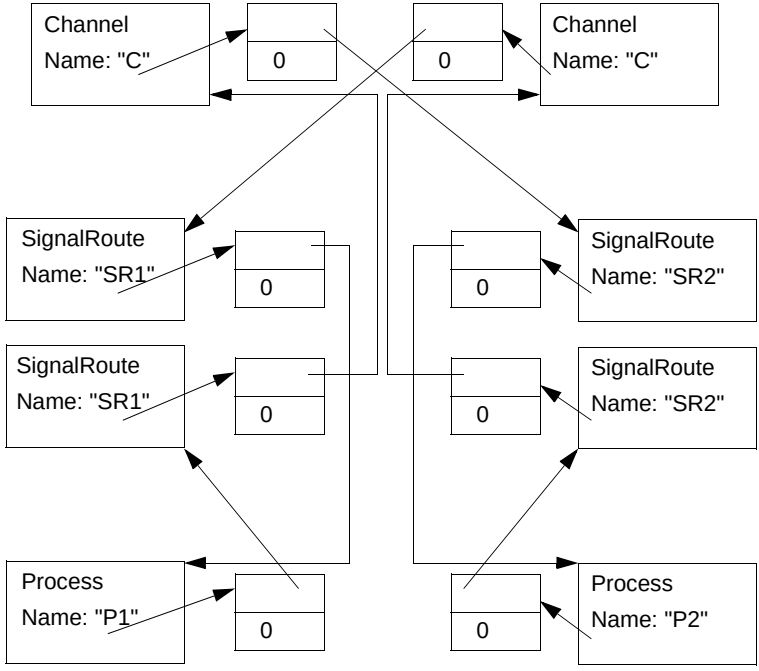


Figure 561: The connection of `ToId` *for the system in Figure 559 and Figure 560*

The Type Concept in SDL-92

The probably most important new feature in SDL-92 is the introduction of the object oriented features, such as TYPE, INHERITS, VIRTUAL, and REDEFINED. Here we start by discussing process types.

For each process type the Cadvanced/Cbasic SDL to C Compiler will generate:

- a `PrsIdNode`
- a `PAD` function
- a `yVDef_ProcessName` struct.

In the `PrsIdNode` there is one component (Super) that will refer to the `PrsIdNode` for the process type inherited by this process type. As sons to a `PrsIdNode`, `IdNodes` for declaration that are common for all instantiation of the process type can be found. Examples of such `IdNodes` are: nodes for variables, formal parameters, signals, timers, procedures, states, newtypes, and syntypes. Any typedefs or help functions for such units are also treated in the process type.

The `PAD` function will be independent of the `PAD` function for a inherited type, each `PAD` function just implementing the action described in its process type.

A `yVDef_ProcessName` struct will on the other hand include all variables and formal parameters from the top of the inheritance chain and downwards. Example:

```
process type P1;
fpar f1 integer;
dcl  d1 integer;
...
endprocess;

process type P2 inherits P1;
fpar f2 integer;
dcl  d2 integer;
...
endprocess;
```

This will generate the following principle yVDef_... structs:

```
typedef struct {
  PROCESS_VARS
  SDL_Integer f1;
  SDL_Integer d1;
} yVDef_P1;
```

```
typedef struct {
  PROCESS_VARS
  SDL_Integer f1;
  SDL_Integer d1;
  SDL_Integer f2;
  SDL_Integer d2;
} yVDef_P2;
```

A pointer to `yVDef_P2` can thus be casted to a pointer to `yVDef_P1`, if only the common component (in `PROCESS_VARS`) or the variables in P1 is to be accessed. This possibility is used every time the `PAD` function for an inherited process type is called.

Each process instantiation will all be implemented as a xPrsIdNode. The `Super` component in such an object refers to the process type that is instantiated. No `PAD` function or yVDef_... struct will be generated. As sons to the `PrsIdNode` for a process instantiation, only such object are inserted that are different in different instantiations. For a process instantiation this is the gates. For other types of information the process instantiation uses the information given for its process type.

A very similar structure when it comes to `IdNodes` generated for block types and block instantiations are used by the code generator. There will be a `BlockIdNode` for both a block type and for a block instantiation. As sons to a block type, nodes that are the same in each block instantiation can be found (example: signal, newtype, syntype, block type, process type, procedure). As sons to a block instantiation, nodes that are needs to be represented in each block instantiation can be found (example: block instantiation, process instantiation, channel, signal route, gate, remote definitions).

Note:

A block or process (according to SDL-88), that is contained in a block type or a system type, is translated as if it was a type and instantiation at the same place.

A way to look at the structure of `IdNodes` in a particular system is to use the command Symboltable in the monitor system. This command prints the `IdNode` structure as an indented list of objects.

Allocating Dynamic Memory

Introduction

This section deals with the allocation and deallocation of dynamic memory.

> **Note:**
>
> This information is only valid when the Master Library is used. The OS integrations might have different strategies for memory allocation.

Information is provided about the following topics:

- Explanation about how dynamic memory is allocated and reused (deallocation and avail lists)
- How to estimate the total need of dynamic memory for an application.

Dynamic memory is used for a number of objects in a run-time model for applications generated by the Cadvanced/Cbasic SDL to C Compiler. These objects are:

- Process instances
- Signal and timer instances
- Procedure instances
- Charstring, Octet_string, Bit_string, and Object_identifer variables and variables of String, Bag, general Array, and general Powerset types.
- Variables of other user-defined data types, where the user has decided to use dynamic memory.

To help to estimate the need for memory for an application we will give information about the size of these objects and about how many of the objects are created. The size information given is true for generated applications, that is, ones that do not, for example, contain the monitor. The type definitions given are stripped of components that will not be part of an application. The full definitions may be found in the file `scttypes.h`.

Processes

Each process instance is represented by two structs that will be allocated on the heap. In `scttypes.h` the type `xLocalPIdRec` is defined and in generated code `yVDef_ProcessName` structs are defined:

```
typedef struct {
   xPrsNode      PrsP;
}  xLocalPIdRec;

typedef struct {
   xPrsNode       Pre;
   xPrsNode       Suc;
   int            RestartAddress;
   xPrdNode       ActivePrd;
   void (*RestartPAD) (xPrsNode  VarP);
#ifndef XNOUSEOFSERVICE
   xSrvNode       ActiveSrv;
   xSrvNode       SrvList;
#endif
   xPrsNode       NextPrs;
   SDL_PId        Self;
   xPrsIdNode     NameNode;
   int            State;
   xSignalNode    Signal;
   xInputPortRec  InputPort;
   SDL_PId        Parent;
   SDL_PId        Offspring;
   int            BlockInstNumber;
   xSignalIdNode  pREPLY_Waited_For;
   xSignalNode    pREPLY_Signal;

   /* variables and formal parameters in the
      process */
} yVDef_ProcessName;
```

To calculate the size of the structs above it is necessary to know more about the components in the structs. The types `xPrsNode`, `xPrdNode`, `xSignalNode`, `xPrsIdNode`, `xStateIdNode`, and `xSignalIdNode` are all pointers, while `SDL_PId` is a struct containing an int and a pointer. The `xInputPortRec` is a struct with two pointers and one int.

This means that it is possible to calculate the size of the `xLocalPIdRec` and the `xPrsRec` struct using the following formulas, if the compiler does not use any strange alignment rules:

$$\text{Size}_{\texttt{xLocalPIdRec}} = \text{Size}_{\text{address}}$$

$$\text{Size}_{\texttt{xPrsRec}} = 16 \cdot \text{Size}_{\text{address}} + 7 \cdot \text{Size}_{\texttt{int}}$$

The size of xPrsRec can be reduced by 2 x sizeof (address) if the code is compiled with the XNOUSEOFSERVICE flag. Then, of course, the SDL concept service cannot be used. The size of `yVDef_ProcessName` is the size of the `xPrsRec` plus the size of the variables and parameters in the process. Any overhead introduced by the C system should also be added. The size of the formal parameter and variables is of course dependent on the declarations in the process. The translation rules for SDL types, both predefined and user defined, can be found in *chapter 56, The Cadvanced/Cbasic SDL to C Compiler*.

For each process instance set in the system the following number of structs of a different kind will be allocated:

- There will be one `xLocalPIdRec` for each process instance created. These structs will not be reused, as they serve as identification of process instances that have existed (see also optimizations below).
- There will be as many `yVDef_ProcessName` structs as the maximum concurrently executing process instances of the process instance set (maximum during the complete execution of the program).

The `yVDef_ProcessName` structs are reused by having an avail list where this struct is placed when the process instance it represents perform a stop action. There is one avail list for each process type. When a process instance should be created, the runtime library first looks at the avail list and reuses an item from the list. Only if the avail list is empty new memory is allocated.

Compilation switch XPRSOPT

If the compilation switch `XPRSOPT` is defined then:

- `xLocalPIdRec`s are reused together with the `xPrsRec`s.
- `xLocalPIdRec`s contain an additional int component.

Services

Services are handled very similar to processes. The following struct type are allocated for each service instance.

```
typedef struct xSrvStruct {
   xSrvNode       NextSrv;
   xPrsNode       ContainerPrs;
   int            RestartAddress;
   xPrdNode       ActivePrd;
   void (*RestartPAD) (xPrsNode  VarP);
   xSrvIdNode     NameNode;
   int            State;
   XSIGTYPE       pREPLY_Waited_For;
   xSignalNode    pREPLY_Signal;
}  xSrvRec;
```

This means that:

$$\text{Size}_{\texttt{xSrvRec}} = 7 \cdot \text{Size}_{\text{address}} + 2 \cdot \text{Size}_{\texttt{int}}$$

The size of `yVDef_ServiceName` is the size of the xSrvRec plus the size of the variables in the service. `yVDef_ServiceName` struct are re-used in the same way as for processes (see previous section).

Signals

Signals are handled in much the same way as processes. A signal instance is represented by one struct (in generated code generated).

```
typedef struct {
   xSignalNode   Pre;
   xSignalNode   Suc;
   int           Prio;
   SDL_PId       Receiver;
   SDL_PId       Sender;
   xIdNode       NameNode;

   /* Signal parameters */
} yPDef_SignalName;
```

This struct type contains one component for each signal parameter. The component types will be the translated version of the SDL types of the parameters.

This means that it is possible can calculate the size of a `xSignalRec`, which is the same as a struct for a signal without parameters, using the following formula:

$$\text{Size}_{\texttt{xSignalRec}} = 5 \cdot \text{Size}_{\text{address}} + 3 \cdot \text{Size}_{\texttt{int}}$$

The size of a `yPDef_SignalName` struct is thus equal to the size of the `xSignalRec` plus the size of the parameters. The translation rules for SDL types, both the predefined and user defined, can be found in *chapter 56, The Cadvanced/Cbasic SDL to C Compiler*.

For each signal type in the system the following number of data areas will be allocated:

- There will be as many `yPDef_SignalName` struct as the maximum number of signals (during the complete execution of the program) of the signal type that are sent but not yet received in an input operation.

The `yPDef_SignalName` struct is reused by having an *avail* list, where the struct is placed when the signal instance they represent is received. The exact point where the signal instance is returned to the avail list is when the transition caused by the signal instance is ended by a nextstate or stop action. There is one avail list for each signal type. When a signal instance should be created, for example during an output operation, the runtime library first looks at the avail list and reuses an item from this list. Only if the avail list is empty new memory is allocated.

Note:

There is one common avail list for all signals without parameters.

Timers

The memory needed for timers can be calculated in the same way as for signals with one exception, each timer contains an extra `SDL_Time` component, i.e. two extra 32-bit integers.

Procedures

Procedures and processes have much in common in terms of memory allocation. A procedure is, during the time it exists from call to return, represented by a struct; the `yVDef_ProcedureName`.

```
typedef struct {
   xPrdIdNode  NameNode;
   xPrdNode    StaticFather;
   xPrdNode    DynamicFather;
   int         RestartAddress;
   int (*RestartPRD) (xPrsNode  VarP);
   xSignalNode pREPLY_Signal;
   int         State;

   /* Formal parameters and variables */
} yVDef_ProcedureName;
```

The struct type contains one component for each formal parameter or variable. The component types will be the translated version of the SDL types of the parameters, except for an IN/OUT parameter which is represented as an address.

The size of the `xPrdRec` struct (which is the same as a procedure without variables and formal parameters) can be calculated using the following formula:

$$\text{Size}_{\texttt{xPrdRec}} = 5 \cdot \text{Size}_{\text{address}} + 2 \cdot \text{Size}_{\texttt{int}}$$

The size of a `yVDef_ProcedureName` struct is the size of the `xPrdRec` plus the size of the formal parameter and variables defined in the procedure. The translation rules for SDL types, both the predefined and user defined can be found in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*.

For each type of procedure in the system the following number of data areas will be allocated:

- There will be as many `yVDef_ProcedureName` structs as the maximum number of concurrent calls (during the complete execution of the program) of the procedure. Concurrent calls occur both when a procedure calls itself recursively within one process instance, and when several process instances of the same process type calls the same procedure during overlapping times.

The `yVDef_ProcedureName` struct is reused by having an avail list, where this two struct is placed when the procedure instance executes a

return action. There is one avail list for each procedure type. When a procedure instance should be created, that is, at a call operation, the runtime library first looks at the avail list and reuses an item in the list. Only if the avail list is empty new memory is allocated.

Data types

The predefined SDL type charstring is implemented as `char *` in C and thus requires dynamic memory allocation. The predefined data types Bit_string, Octet_string, and Object_identifier are also implemented using dynamic memory.

The implementation of the SDL sorts Charstring, Bit_string, Octet_string, and Object_identifier is both flexible in length and all memory can be reused.

The mechanism used to release unused memory is to call the `xFree` function in the file `sctos.c`, which uses the standard function free to release the memory.

Charstrings, Bit_strings, Octet_strings, and Object_identifiers are also handled correctly if they are part of structs or arrays. When, for example, a new value is given to a struct having a charstring component, the old charstring value will be released. For all structured types containing any of these types there will also be a `Free` function that is utilized to release all dynamic memory in the structured variable.

Functions for Allocation and Deallocation

The allocation and deallocation of memory is handled by the functions `xAlloc` and `xFree` in the file `sctos.c`. The functions in this file are used for the adoption of the generated applications to the operating system or hardware. The `sctos.c` file is described in detail in "The sctos.c File" on page 3149.

In generated code and in the run-time library the functions `xAlloc` and `xFree` are used in each situation where memory is needed or can be released. `xAlloc` receives as parameter a requested size in bytes and returns the address to a data area of the requested size. All bytes in the data area are set to zero. `xFree` takes the address of a pointer and returns the data area referenced by the pointer to the pool of free memory. It also sets the pointer to 0.

The xAlloc and xFree functions are usually implemented using some version of the C standard functions for allocation (malloc, calloc) and deallocation (free). Other implementations are of course possible as long as the interface described in the previous section is fulfilled. In a micro controller, for example, it is probably necessary to handle allocation and deallocation directly towards the physical memory.

To prevent memory fragmentation we have used our own avail lists in almost all circumstances. Memory fragmentation is phenomena occurring when a program allocates and de-allocates data areas (of different sizes) in some “random” order. Then small pieces of memory here and there are lost, since their sizes are to small to fit an allocation request. This can lead to a slowly increasing demand for memory for the application.

Note that deallocation of memory is only used for data types. More specific it is used for variables of type:

- Charstring
- Octet_string
- Bit_string
- Object_identifier
- Types created by String (not #STRING) and Bag generator
- Types created by Array generator, if the index type is such that an array in C cannot be used. (General array)
- Types created by Powerset generator, if the component type has the has property as for the index type in general arrays.

This means that if variables of the above mentioned types are not used and the user has not introduced the need for deallocation of memory himself, no memory deallocation will occur. In this case it is of course unnecessary to implement the xFree function.

It is easy to trace the need for dynamic memory. As all memory allocation is carried out through the xAlloc function and this function is available in source code (in sctos.c), it is only necessary to introduce whatever count statements or printout statements that are appropriate.

Compilation Switches

The compilation switches are used to decide the properties of the Master Library and the generated C code. Both in the library and in generated code `#ifdefs` are used to include or exclude parts of the code.

The switches that are used can be divided into four groups.

1. Switches defining properties of the compiler.
2. Switches defining a library version.
3. Switches defining a property of a library version.
4. Switches defining the implementation of a property.

The first group will be discussed in "Adaptation to Compilers" on page 3147.

The following switches define the library version:

Switch	Corresponds to Library
`SCTDEBCOM`	*Simulation*
`SCTDEBCLCOM`	*RealTimeSimulation*
`SCTAPPLCLENV`	*Application*
`SCTDEBCLENVCOM`	*ApplicationDebug* (Simulation with environment)
`SCTPERFSIM`	*PerformanceSimulation* (Library with simulated time, no environment functions, no monitor.)

The definition of the properties of these libraries can be found in `scttypes.h` and will be discussed below. Each library version is specified by the switches in the group property switches that it defines.

New library versions, containing other combinations of property switches, can easily be defined by introducing new library definitions in the `scttypes.h` file.

The property switches discussed below can be used to form library versions. If not stated otherwise for a certain property, all code, variables, struct components, and so on, are either included or excluded using conditional compiling (`#ifdef`), depending on whether the property is used or not.

This means, for example, that all code for the monitor interface will be removed in an application not using the monitor, which makes the application both smaller and faster.

Description of Compilation Switches

XCLOCK

If this compilation switch is not defined then simulated time is used, otherwise the system time is connected to a real clock, via the `sctos.c` function `SDL_Clock`.

XCALENDARCLOCK

This is the same as `XCLOCK` (it will actually define `XCLOCK`), except that if `XCLOCK` is used, time will be zero at system start up, while if `XCALENDARCLOCK` is used, time will be whatever the clock returns at system start up.

XPMCOMM

Define this compilation switch if the application should be able to communicate with signals via the SDL Suite communication mechanism. This facility is used to accomplish communicating simulations and simulations communicating with, for example, user interfaces.

XITEXCOMM

This switch should be defined if a generated simulator should be able to communicate with a TTCN simulator.

XENV

If this compilation switch is defined the environment functions `xInitEnv`, `xCloseEnv`, `xInEnv`, and `xOutEnv` will be called at appropriate places.

XTENV

This is the same as `XENV` (it will actually define `XENV`), except that xInEnv should return a time value which is the next time it should be called (a value of type `SDL_Time`). The main loop will call `xInEnv` at

the first possible occasion after the specified time has expired, or when the SDL system becomes idle.

XENV_CONFORM_2_3

This switch make signals using a compatible data structure as in SDT 2.3. This means that an extra and unnecessary component `yVarP` is inserted in each signal.

XSIGLOG

This facility makes it possible for a user to implement his own log of the major events in the system. This compilation switch is normally not defined. By defining this switch, each output of a signal, i.e. each call of the function `SDL_Output`, will result in a call of the function `xSignalLog`. Each time a transition is started, the function `xProcessLog` will be called.

These functions have the following prototypes:

```
extern void xSignalLog
  (xSignalNode  Signal,
   int          NrOfReceivers,
   xIdNode    * Path,
   int          PathLength);

extern void xProcessLog
  (xPrsNode P);
```

which are included in `scttypes.h` if `XSIGLOG` is defined.

`Signal` will be a pointer to the data area representing the signal instance.

`NrOfReceivers` will indicate the success of the output according to the following table:

NrOfReceivers	Output Statement Contents
-1:	A TO clause, but no path of channels and signal routes were found between the sender and the receiver.
0:	No TO clause, and no possible receivers were found in the search for receivers.

NrOfReceivers	Output Statement Contents
1:	If the output statement contains a TO clause, a path of channels and signal routes was found between the sender and the receiver. If the output statement contains no TO clause, exactly one possible receiver was found in the search for receivers. The output was thus successful. The only error situation that still might be present is if an output with a TO clause is directed to a process instance that is stopped.

The third parameter, `Path`, is an array of pointer to `IdNodes`, where `Path[0]` refers to the `IdNode` for the sending process, `Path[1]` refers to the first signal route (or channel) in the path between the sender and the receiver, and so on, until `Path[PathLength]` which refers to the `IdNode` for the receiving process.

The parameter `P` in the `xProcessLog` function will refer to the process just about to start executing.

The fourth parameter, `PathLength`, represents thus the number of components in the `Path` array that are used to represent the path for the signal sent in the output. If the signal is sent to or from the environment, either `Path[0]` or `Path[PathLength]` will refer to `xEnvId`, that is to the `IdNode` for the environment process.

In the implementation of the `xSignalLog` and `xProcessLog` functions which should be provided by the user, the user has full freedom to use the information provided by the parameters in any suitable way, except that it is not possible to change the contents of the signal instance. The functions are provided to make it possible for a user to implement a simple log facility in environments where standard IO is not provided, or where the monitor system is too slow or too large to fit. A suitable implementation can be found in the file `sctenv.c`

XTRACE

If this compilation switch is defined, traces of the execution can be printed.

This facility is normally used together with the monitor, but can also be used without the monitor. The file `stdout` must of course be available for printing.

Setting trace values must, without the monitor, be performed in included C code, as the monitor interface is excluded. The trace components are called `Trace_Default` and can be found in `IdNodes` representing system, blocks, and processes, and in the struct `xPrsRec` used to represent a process instance. The values stored in these components are the values given in the Set-Trace command in the monitor. The value undefined is represented by -1.

When the monitor is excluded all trace values will be undefined at startup, except for the system which has trace value 0. This means that no trace is active at start up.

Example 512 ––

Suitable statements to set trace values in C code:

```
xSystemId->Trace_Default = value;
   /* System trace */
xPrsN_ProcessName->Trace_Default = value;
   /* Process type trace */
PId_Var.LocalPId->PrsP->NameNode->Trace_Default =
  value
   /* Process type trace */
PId_Var.LocalPId->PrsP->Trace_Default = value;
   /* Process instance trace */
```

`PId_Var` is assumed to be a variable of type `PId`.

> **Note:**
>
> Note that the variable `xPrsN_ProcessName` is declared, and therefore only available, in the file containing the block where the process is defined (and in files representing processes contained in the block).

––

XGRTRACE

If this compilation switch is defined it is possible for a simulation to communicate with the Organizer and the SDL Editor to highlight SDL symbols in the graphical representation.

This feature is used together with the monitor to implement graphical trace and commands like Show-Next-Symbol and Show-Previous-Symbol. It is possible to use graphical trace without the monitor in the same way as the ordinary trace (substitute `Trace_Default` with `GRTrace` in the description above). However the graphical trace is synchronized which means that the speed of the application is dramatically reduced.

XCTRACE

Defining this compilation switch makes information available to the monitor about where in the source C code the execution is currently suspended. This facility, which is used together with the monitor, makes it possible to implement the monitor command Show-C-Line-Number.

XMONITOR

If this compilation switch is defined, the monitor system is included in the generated application.

XCOVERAGE

This compilation switch makes it possible to generate coverage tables. It should be used together with `XMONITOR`.

MAX_READ_LENGTH

This macro controls the length of the `char *` buffers used to read values of SDL sorts. A typical usage is when the monitor commands Assign-Value is entered. If large data types are used, it is possible to redefine the sizes of the buffers from their default size (10000 bytes) to something more appropriate.

XSIMULATORUI

This compilation switch should be defined if the generated simulator is to be executed from the Graphical User Interface to the simulator monitor.

XMSCE

This compilation switch should be defined if the generated simulator should be able to generate Message Sequence Charts.

XSDLENVUI

This compilation switch should be defined if it should be possible to start and communicate with a user interface (or another application) from the simulation. This feature should be used together with the monitor and will define the switch XPMCOMM (see also this switch).

XNOMAIN

When this compilation switch is defined the functions main and xMainLoop are removed using conditional compiling. This feature is intended to be used when a generated SDL application should be part of an already existing application, that is when the SDL system implements a new function in an existing environment. The following functions are available for the user to implement scheduling of SDL actions:

```
extern void xMainInit(
  void (*Init_System) (void)
#ifdef XCONNECTPM
 ,int argc,
  char *argv[]
#endif
  );

#ifdef XNOMAIN
extern void SDL_Execute (void);

extern int SDL_Transition_Prio (void);

extern void SDL_OutputTimer (void);

extern int SDL_Timer_Prio (void);

extern SDL_Time SDL_Timer_Time (void);
#endif
```

The behavior of these functions are as follows:

xMainInit: This function should be called to initialize the SDL system before any other function in the runtime library is called. An appropriate way to call xMainInit is:

```
#ifdef XCONNECTPM
xMainInit(yInit, argc, argv);
#else
xMainInit(yInit);
#endif
```

The compilation switch XCONNECTPM will be defined if the any switch that requires communication via the SDL Suite communication mechanism is defined (XPMCOMM or XGRTRACE).

SDL_Execute: This function will execute one transition by the process instance first in the ready queue.

Before calling this function it must be checked that there really is at least one process instance in the ready queue. This test can be performed using the function SDL_Transition_Prio discussed below.

SDL_Transition_Prio: This function returns the priority of the process first in the ready queue (if signal priorities are used it is the priority of the signal that has caused the transition by the actual process instance). If the ready queue is empty, -1 is returned.

SDL_OutputTimer: This function will execute one timer output and may only be called if there is a timer ready to perform a timer output. This test can be performed with either SDL_Timer_Prio or SDL_Timer_Time described below.

SDL_Timer_Prio: This function returns the priority of the timer first in the timer queue if the timer time has expired for this timer. That is, if Now is greater than or equal to the time given in the Set statement for the timer.

If the timer queue is empty or the timer time for the first timer has not expired, -1 will be returned.

If signal priorities are used, the priority returned is the priority assigned to the timer type (in the timer definition) or the default timer priority; while if process priorities are used the priority returned is the priority of the process that has set the timer.

SDL_Timer_Time: This function returns the time given in the set statement for the first timer in the timer queue. If the timer queue is empty, the largest possible time value (xSysD.xMaxTime) is returned.

Depending on how the SDL system is integrated in an existing environment it might be possible to also use the monitor system. In that case the function xCheckMonitors should be called to execute monitor commands.

```
extern void xCheckMonitors (void);
```

To give some idea of how to use the functions discussed above, an example reflecting the way the internal scheduler in the runtime library works is given below:

Example 513 ----------

```
  while (1) {
#ifdef XMONITOR
    xCheckMonitors();
#endif
    if ( SDL_Timer_Prio() >= 0 )
      SDL_OutputTimer();
    else if ( SDL_Transition_Prio() >= 0 )
      SDL_Execute();
  }
```

XMAIN_NAME

Sometimes when integrating generated application or simulations in larger environments the main function can be useful but cannot have the name `main`. This name can be changed to something else by defining the macro `XMAIN_NAME`. The main function came be found in the file `sctsdl.c`.

XSIGPRIO

The `XSIGPRIO` compilation switch defines that priorities on signals (set in Output statements) should be used. This switch and the three other switches for priorities given below are, of course, mutually exclusive.

A signal priority is specified with a priority directive (see "Assigning Priorities – Directive #PRIO" on page 2739 in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*, that is by a comment with the following outline:

```
/*#PRIO 5 */.
```

A priority can be assigned to a signal instance in an output statement by putting a #PRIO directive **last in the output symbol**. In SDL/PR it is possible to put the #PRIO directive both immediately before and immediately after the semicolon ending the output statement. The Cadvanced/Cbasic SDL to C Compiler will first look for #PRIO directives in the output statement. If no directive is found there it will look in the signal definition for the signal for a priority directive. A #PRIO direc-

tive should be placed directly **before** the comma or semicolon ending the definition of the signal.

Example 514 ----------

```
SIGNAL
  S1 /*#PRIO 3 */,
  S2 (Integer) /*#PRIO 5 */;
```

If no priority directive is found in the output symbol or in the definition of the signal, the default value for signal priority is used. This value is 100. Timers can be assigned priorities in timer definitions in the same way as signals in signal definitions.

The signal priorities will be used to sort the input port of process instances in priority order, so that the signal with highest priority (lowest priority value) is at the first position. Two signals with same priority are placed in the order they arrive. The priority of the signal that can cause the next transition by a process instance is used to sort the ready queue in priority order, so that the process with a signal of highest priority is first. With equal priority, the processes are placed in the order they are inserted into the ready queue. If a continuous signal caused a processes to be inserted into the ready queue, it is the priority of the continuous signal that will be used as signal priority for this “signal”.

Note that a start transition also have a “signal priority”. This is by default also 100 and is set by the macro `xDefaultPrioCreate` described below.

Caution!

Signal priority is not included in SDL according to ITU Recommendation Z.100, and that sorting the signals in the input port of a process instance according to priorities is a direct violation of the SDL standard. This feature is however included for users that need such a behavior to implement their applications.

XPRSPRIO

This compilation switch defines that process priorities should be used. For more information see chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*, section Assigning Priorities - Directive #PRIO.

XSIGPRSPRIO

This compilation switch defines that priorities on signals should be used as first key for sorting in priority order, and process priorities should be used as second key.

XPRSSIGPRIO

This compilation switch defines that process priorities should be used as first key for sorting in priority order, and priorities on signals should be used as second key.

xDefaultPrio...

It is possible to redefine the default priorities for processes, signals, timer signals, continuous signals and start-up signals by defining the symbols below to appropriate values. The default value for these defaults are 100.

```
xDefaultPrioProcess
xDefaultPrioSignal
xDefaultPrioTimerSignal
xDefaultPrioContSignal
xDefaultPrioCreate
```

XOPT

This compilation switch will turn on full optimization (except XOPTCHAN), that is, it will define the following switches:

XOPTSIGPARA	XOPTDCL
XOPTFPAR	XOPTSTRUCT
XOPTLIT	XOPTSORT

For more information, see these switches below. The XOPT switches should not be used together with the monitor.

XOPTSIGPARA

In the symbol table tree (see section "Symbol Table Tree Structure" on page 3028) there will be one node for each parameter to a signal. These nodes are not necessary in an application and can be removed by defining the compilation switch XOPTSIGPARA.

XOPTDCL

There will be a `VarIdNode` in the symbol table tree for each variable declared in processes, procedures, or operator diagram. These nodes are not used in an application (without the monitor) and can be removed by defining the compilation switch `XOPTDCL`.

XOPTFPAR

There will be a `VarIdNode` in the symbol table tree for each formal parameter in a processes, procedures, or operator diagram. These node are not used in an application and may be removed by defining the compilation switch `XOPTFPAR`.

XOPTSTRUCT

For each component in an SDL struct there will be one `VarIdNode` defining the properties of this component. These `VarIdNodes` are not used in an application and can be removed by defining the compilation switch `XOPTSTRUCT`.

XOPTLIT

For each literal in a newtype that will be translated to an enum type, there will be an `LitIdNode` representing the literal. These nodes will not be used in an application and can be removed by defining the compilation switch `XOPTLIT`.

XOPTSORT

Each newtype and syntype, including the SDL standard types, will be represented by an `SortIdNode`. These nodes are not used in an application if all the other `XOPT...` mentioned above are defined.

XNOUSEOFREAL

Defining this compilation switch will remove all occurrences of C `float` and `double` types, and means for example that the SDL type Real is no longer available.

This switch is intended to be used in situations when it is important to save space, to see to that the library functions for floating type operations are not necessary to load. It cannot handle situations when the user includes floating type operations in C code, for example #CODE directives. Another consideration is if BasicCTypes.pr, or other ADTs, are

included in the system. If so, it is required that types dependent on SDL Real be removed from these packages.

XNOUSEOFOBJECTIDENTIFER

Defining this switch will remove all code for the SDL predefined sort Object_identifier.

XNOUSEOFOCTETBITSTRING

Defining this switch will remove all code for the SDL predefined sorts Bit_string, Octet, and Octet_string.

Special consideration needs to be taken if BasicCTypes.pr, or other ADTs, are included in the system. If so, it is required that types dependent on these types be removed from these packages.

XNOUSEOFEXPORT

By defining this switch the user states that he is not going to use the export - import concept in SDL.

> **Caution!**
>
> An attempt to perform an import operation when XNOUSEOFEXPORT is defined will result in a compilation error, as the function `xGetExportAddr` is not defined.

XNOUSEOFSERVICE

This compilation switch can be defined to save space, both in data and in the size of the kernel, if the SDL concept service is not used. If services are used and this switch is defined, there will be compilation errors (probably many!), when the generated code is compiled.

XPRSOPT

Section "Create and Stop Operations" on page 3085 describes how `xLocalPIdRec` structs are allocated for each created process instance, and how these structs are used to represent process instances even after they have performed stop actions. This method for handling `xLocalPIdRecs` is required to be able to detect when a signal is sent to a process instance that has performed a stop operation.

In an application that is going to run for a “long” period of time and that uses dynamic processes instances, this way of handling `xLocalPIdRecs` will eventually lead to no memory being available.

By defining the compilation switch `XPRSOPT`, the memory for the `xLocalPIdRecs` will be reused together the `yVDef_ProcessName` structs. This has two consequences:

1. The need for memory will not increase due to the use of dynamic processes (the memory need depends on the maximum number of concurrent instances).
2. It will no longer be possible to always find the situation when a signal is sent to a process instance that has performed a stop action.

More precisely, if we have a PId variable that refers to a process instance which performs a stop operation and after that a create operation (on the same process instance set) is performed where the same data area is reused, then the PId variable will now refer to the new process instance.

This means, for example, that signals intended for the old instance will be sent to the new instance. Note that it is still possible to detect signal sending to processes in the avail list even if `XPRSOPT` is defined.

XOPTCHAN

This switch can be used to remove all information about the paths of channels and signal routes in the system. The following memory optimization will take place:

- The two `ChannelIdNodes` for each channel, signal route, and gate are removed.
- The `ToId` component in the `xPrsIdNodes` representing processes is removed.
- A number of functions in the library (`sctsdl.c`) are no longer needed and are removed.

When the information about channels, signal routes, and gates is not present two types of calculations can no longer be performed:

1. To check if there is a path of channels and signal routes between the sender and the receiver in an OUTPUT statement with a TO clause.

This is no problem as this is just an error test that we probably do not want to be performed in an application.

2. To calculate the receiver in an OUTPUT without TO clause, if the Cadvanced/Cbasic SDL to C Compiler has not performed this calculation at generate time (see *"Calculation of Receiver in Outputs" on page 2655 in chapter 56, The Cadvanced/Cbasic SDL to C Compiler*). This is more serious, as it means that **OUTPUT without TO cannot always be used. The restrictions are:**

   – No outputs without to in process types, or in process in block or system types.
   – No outputs without to, designated to a process in a SEPARATE unit.

> **Caution!**
>
> If the `XOPTCHAN` switch is defined and still OUTPUT without TO clause are used (which the Cadvanced/Cbasic SDL to C Compiler cannot optimize), there will be a C compilation error saying that the name `xNotDefPId` is not defined.

In an ordinary SDL system OUTPUTs without TO must be used to start up the communication between different parts of the system, as there is no other way in SDL to distribute the PId values needed for OUTPUTs with TO.

This problem is solved if the Cadvanced/Cbasic SDL to C Compiler can calculate the receiver. Otherwise the data type `PIdList` in the library of abstract data types is intended to solve this problem. It is described in chapter 62, *The ADT Library*. When this data type is used, global PId literals my be introduced, implemented as SDL synonyms. These literals can then be used to utilize OUTPUT statements with TO clauses from the very beginning.

X_LONG_INT

The SDL sort Integer is translated to int in C. To translate the Integer sort to long int instead, just define the compilation switch `X_LONG_INT`.

XENVSIGNALLIMIT

If this switch is defined, only a limited number of signals will be stored in the input port of the Env function. The limit is equal to the value defined for `XENVSIGNALLIMIT` and is normally set to 20.

XEALL

This switch will define all error handling switches (XE...) and `XASSERT` given below.

XECREATE

This switch will report if the initial number of instances of a process type is greater than the maximum number.

XECSOP

This switch will report error situations in ADT operator.

XEDECISION

This switch will report if no path out from a Decision is found.

XEEXPORT

This switch will report errors during Import actions.

XEFIXOF

This switch will report overflow when an SDL Real value is converted to an SDL Integer value using the operator Fix.

XEINDEX

This switch will report value out of range for array index.

XEINTDIV

This switch will report division by zero in an integer division.

XEOUTPUT

This switch will report errors during Output operations.

XERANGE

This switch will report range errors when a value is assigned to a variable of a sort containing range conditions.

XEREALDIV

This switch will report division by zero in a real division.

XEVIEW

This switch will report errors in View operations

XECHOICE

This switch will turn on error reports when accessing non-active choice components.

XEOPTIONAL

This switch will turn on error reports when accessing non-present optional struct components.

XEUNION

This switch will turn on error reports when accessing non-active union components.

XEREF, XEOWN

These switches turn on error checking on pointers (generator Ref and Own).

XASSERT

By defining this switch the possibility to define user assertions which is described in "Assertions" on page 2196 in chapter 49, *The SDL Simulator*.

XTRACHANNELSTOENV

When using partitioning of a system a problem during the redirection of channels is that the number of channels going to the environment is not known at code generation time, which means that the size of the data

area used for the connections is not known. This problem is solved in two ways.

Either the function handling redirections allocates more memory, which is the default, or the user specifies how many channels that will be redirected (which could be difficult to compute, but will lead to less need of memory).

In the first case (allocation of more memory) the macros:

```
#define XTRACHANNELSTOENV  0
#define XTRACHANNELLIST
```

should be defined like above. This is the standard in scttypes.h. If the user wants to specify the number of channels himself then

```
#define XTRACHANNELSTOENV  10
#define XTRACHANNELLIST    ,0,0,0,0,0,0,0,0,0,0
```

i.e. XTRACHANNELSTOENV should be the number of channels, while XTRACHANNELLIST should be a list of that many zeros.

XDEBUG_LABEL

It is for debugging purposes sometimes of interest to introduce extra labels. The macro XDEBUG_LABEL is inserted in the code for each input symbol. As macro parameter it has a name which is the name of the state concatenated with an underscore concatenated with the signal name.

Example 515 ––

```
state State1; input Sig1;
state State2; input *;
state *; input Sig2;
```

In the generated code for these input statements the following macros will be found:

```
XDEBUG_LABEL(State1_Sig1)
XDEBUG_LABEL(State2_ASTERISK)
XDEBUG_LABEL(ASTERISK_Sig2)
```

A suitable macro definition to introduce label would be:

```
#define XDEBUG_LABEL(L)  L: ;
```

To use these label the usage of SDL must be restricted in one area. The same state may not receive two different signals with the same name! This is allowed and handled by the SDL Suite. The signal have to be

defined at different block or system level and the outermost signal must be referenced with a qualifier.

XCONST, XCONST_COMP

Using these compilation switches most of the memory used for the `IdStructs` can be moved from RAM to ROM. This depends of course on the compiler and what properties it has.

The following macro definitions can be inserted:

```
#define XCONST const
#define XCONST_COMP const
```

This will introduce **const** in the declaration of most of the `IdStructs`. It is then up to the compiler to handle const.

The XCONST_COMP macro is used to introduce const on components within a struct definition. This is necessary for some compilers to accept const on the struct as such.

If `const` is successfully introduced, there is a lot of RAM memory that will be saved, as probably 90% of the data area for `IdStructs` can be made `const`.

XAVL_TIMER_QUEUE

This compilation switch changes the timer queue data structure from the default sorted double linked list to an AVL tree. Each node in the tree represents a specific TimerTime and holds all timers scheduled to fire at that time. Insertions grow logarithmically with the total number of scheduled timers, instead of linearly as is the case with the ordered list. For models with a large number of timers the AVL tree can offer a significant performance boost. The AVL tree requires slightly more memory and is not recommended for models with few timers.

Compilation Switches – Summary

The property switches are in principle independent, except for the relations given in the descriptions above, and it should always be possible to any combination.

The number of combinations is, however, so huge that it is impossible for us to even compile all combinations. If you happen to form a combination that does not work, please let us know, so that we either can correct the code, or, if that is not possible, publish a warning against that combination.

The switches defining a standard library version will define the following property switches:

SCTDEBCOM	**SCTDEBCLCOM**
XPRSPRIO XPARTITION XEALL XMONITOR XTRACE XCTRACE XMSCE XCOVERAGE XGRTRACE XPMCOMM XSDLENVUI XITEXCOMM XSIMULATORUI	XCLOCK XPRSPRIO XPARTITION XEALL XMONITOR XTRACE XCTRACE XMSCE XCOVERAGE XGRTRACE XPMCOMM XSDLENVUI XSIMULATORUI
SCTAPPLCLENV	**SCTDEBCLENVCOM**
XCALENDARCLOCK XENV XPRSPRIO XOPT XPRSOPT	XCALENDARCLOCK XPRSPRIO XPARTITION XENV XPRSOPT XEALL XMONITOR XTRACE XCTRACE XMSCE XCOVERAGE XGRTRACE XPMCOMM XSDLENVUI XSIMULATORUI
SCTPERFSIM	
XEALL XPRSPRIO	

The lowest layer of switches (that handle the implementation details) are set up using the three layers above. These switches will not be discussed here. Please refer to the source code files `scttypes.h` and `sctsdl.c` for more details.

Creating a New Library

Caution!

If you create new versions of the library, make sure that the library and the generated code are compiled with the same compilation switches. If not, you might experience any type of strange behavior in the generated application!

This section describes how to generate a new library. The following topics are covered:

- The **directory structure** for source and object code.
- The **`sdtsct.knl`** file, which determines what libraries the Analyzer knows about, that is, what libraries that will be shown when Generate-Options is selected.
- The **`comp.opt`** file and the **`makeoptions`** (**`make.opt` in Windows**) file, which determines the properties of an object code library.
- The make file **`Makefile`**, which includes the `makeoptions (make.opt)` file and generates a new object code library with the properties given by the included `makeoptions (make.opt)` file.
- The relations with the generated make files for SDL systems will also be discussed.

Directory Structure

The structure of files and directories used for the Cadvanced/Cbasic SDL to C Compiler libraries is shown in Figure 562 The directory `sdtdir` is in the installation:

```
<installation directory>/sdt/sdtdir/<machine
dependent dir>
```

where <*machine dependent dir*> is for example `sunos5sdtdir` on SunOS 5 and `wini386` in Windows. (**In Windows**, `/` should be replaced by `\` in the path above.)

This directory is here called sdtdir and is in UNIX normally referred to by the environment variable `sdtdir`.

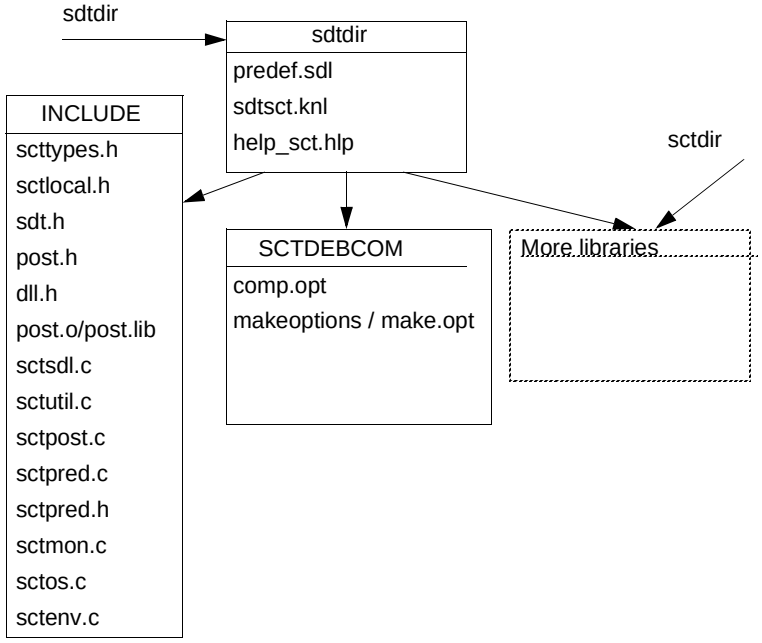


Figure 562: Directory structure

"sctdir" is a reference to an object library and is usually setup as a parameter in the call to make. It can also be an environment variable.

In the `sdtdir` directory three important files are found:

3. `predef.sdl` contains the definition of the predefined sorts in SDL.
4. `sdtsct.knl` contains a list of the available libraries that can be used together with code generated by the Cadvanced/Cbasic SDL to C Compiler.
5. `help_sct.hlp` contains the help information that can be obtained using the monitor command help.

The file `predef.sdl` is read by the SDL Analyzer during analysis, while the file `sdtsct.knl` is used to present the available libraries in the Make dialog in the Organizer (see "Make" on page 120 in chapter 2, *The Organizer*).

In the INCLUDE directory, there are two important groups of files:

1. The source code files for the runtime library:
   scttypes.h, sctlocal.h, sctsdl.c, sctutil.c, sctpost.c, sctpred.h, sctpred.c, sctmon.c, sctos.c and sctenv.c
2. The files necessary to include communication with other SCT applications: post.h, post.o (post64.o for solaris 64-bit, post.lib **in Windows**), sdt.h, itex.h.

In parallel with the INCLUDE directory there are a number of directories for libraries in object form. The SCTDEBCOM directory in Figure 562 is an example of such a directory. Each of these directories will contain three files: comp.opt, makeoptions (make.opt **in Windows**).

The comp.opt file determines the contents of the generated makefile and how make is called. For more details see below.

The makeoptions (make.opt) file describes the properties of the library, such as the compiler used, compiler options, linker options, and so on.

To guarantee the consistency of, for example, compilation flags between the SDL system and the kernel, the makeoptions (make.opt) file is used both by the make file compiling the library (Makefile) and by the generated make files used to compile the generated SDL system. Non-consistency in this sense between the library and the SDL system will make the result unpredictable.

File sdtsct.knl

The sdtsct.knl file describes which libraries that are available. This is presented by the Organizer in the *Make* dialog see "Make" on page 120 in chapter 2, *The Organizer*). The sdtsct.knl file has the following structure. Each available library is described on a line of its own. Such a line should first contain the name of the library (the name presented in the dialog), then the path to the directory containing the library, and last a comment up to end of line.

The path to the library can either be the complete path or a relative path. A relative path is relative to the environment variable sdtdir **(on UNIX)** or SDTDIR **(in Windows)**, if that variable is defined. Otherwise it is relative to the SDL Suite installation.

Example 516 ----------

```
Simulation           SCTDEBCOM
RealTimeSimulation   SCTDEBCLCOM
Application          SCTAPPLCLENV
ApplicationDebug     /util/sct/SCTDEBCLENVCOM

ApplicationDebug     x:\sdt\sdtdir\dbclecom
MyTestLibrary        ..\testlib\dbcom
```

Not that the two last lines is examples for Windows, while the fourth line is for UNIX.

The Organizer will look for an `sdtsct.knl` file first in the directory the SDL Suite is started from, then in the home directory for the user, and then in the directory referenced by the environment variable `sdtdir` (`SDTDIR`) if it is defined, and in the directory where the SDL Suite was installed.

File Makefile

In each directory that contains a library version there is a `Makefile` that can "make" the library. To create a new library after an update of the source code, change directory to the directory for the library and execute the `Makefile`. The `Makefile` uses the `makeoptions (make.opt)` file in the directory to get the correct compilation switches and other relevant information.

Caution!

Do not generate and test libraries in the installation directory structure. Create an appropriate copy.

Note:

The environment variables (if used) `sdtdir (SDTDIR)` and `sctdir (SCTDIR)` need not necessarily refer to directories in the installation directory. Any directory containing the relevant files may be used.

File comp.opt

This file determines the details of the generated make files, and the command issued to execute the makefile. A comp.opt file contains zero,

one or more initial lines starting with a #. These lines are treated as comments. After that it contains five lines of essential data.

- Line 1: How to include the makeoptions (make.opt) file
- Line 2: Compile script
- Line 3: Link script
- Line 4: Command to run make
- Line 5: How to build a library (archive). Used for coders/decoders.

On each of these lines % codes can be used to insert specific information.

On all five lines:

```
%n : newline
%t : tab
%d : target directory
%s : source directory
%k : kernel directory
%f : base name of generated executable (no path, no
     file extension). NOT on line 2 or 5.
```

On line 2, the compile script:

```
%c : c file in compile script
%C : c file in compile script, without extension
%o : resulting object file in compile script
```

On line 3, the link script:

```
%o : list of all object files in link script
%O : list of all object files in link script, with
     \ followed by newline between files
%e : executable file in link script
```

On line 4, the make command:

```
%m : name of generated makefile
```

On line 5, the archive command:

```
%o : list of object files, i.e. $(sctCODER_OBJS).
%a : the archive file, i.e.
     libstcoder$(sctLIBEXTENSION)
```

Example 517: comp.opt file for UNIX ----------

```
# makefile for unix make
include $(sctdir)/makeoptions
%t$(sctCC) $(sctCPPFLAGS) $(sctCCFLAGS) $(sctIFDEF) %c -o %o
%t$(sctLD) $(sctLDFLAGS) %o -o %e
make -f %m sctdir=%k
%t$(sctAR) $(sctARFLAGS) %a %o
```

File makeoptions / make.opt

This file has the following structure:

Example **on UNIX:**

```
# #
sctLIBNAME     = Simulation
sctIFDEF       = -DSCTDEBCOM
sctEXTENSION   = _smd.sct
sctOEXTENSION  = _smd.o
sctLIBEXTENSION=  _smd.a
sctKERNEL      = $(sctdir)/../INCLUDE
sctCODERDIR    = $(sctdir)/../coder

#Compiling, linking
sctCC          = cc
sctCODERFLAGS  = -I$(sctCODERDIR)
sctCPPFLAGS    = -I. -I$(sctKERNEL) $(sctCODERFLAGS)
 $(sctCOMPFLAGS) $(sctUSERDEFS)
sctCCFLAGS     = -c -Xc
sctLD          = cc
sctLDFLAGS     =
sctAR          = ar
sctARFLAGS     = rcu

all : default

# below this point there are a large number of
# compilation rules for compiling the Master Library
# and the Coder library (used for encoding/decoding)
# The following name of any importance are defined:

sctLINKKERNEL  =
sctLINKKERNELDEP =
sctLINKCODERLIB =
sctLINKCODERLIBDEP =
```

The information to the right of the equal signs should be seen as an example. These environment variables set in the `makeoptions` (`make.opt`) file should specify:

- **`sctLIBNAME`**. This is only used by the `Makefile` to report what it is doing.
- **`sctIFDEF`**. This variable should specify what compilation switches, among those defined by the Cadvanced/Cbasic SDL to C Compiler system, that should be used. Usually there is one switch defining the library version.

- **sctEXTENSION**. This is used to determine the file extension of the executable files.
- **sctOEXTENSION**. This is used to determine the file extension of the object files.
- **sctLIBEXTENSION**. The extension of the archive/library
- **sctKERNEL**. Directory of Master Library source code.
- **sctCODERDIR**. The directory for the source code of the coders/decoders.
- **sctCC**. This defines the compiler to be used.
- **sctCODERFLAGS**. Compilation options needed to compile the coder/decoder files
- **sctCPPFLAGS**. This variable should give the compilation flag necessary to specify where the C preprocessor can find the include files `scttypes.h`, `sctlocal.h`, `sctpred.h`, `sdt.h`, and `post.h`.
- **sctCCFLAGS**. This should specify other compiler flags that should be used, as for example -g (Sun cc) or -v (Borland bcc32) for debug information, -O for optimization.
- **sctLD**. This defines the linker to be used.
- **sctLDFLAGS**. This should specify other flags that should be used in the link operation.
- **sctAR**. The archive application
- **sctARFLAGS**. Flags to sctAR.
- **sctLINKKERNEL**. This variable should specify the `.o` files for the Master Library source files. It will be used in the link command in the generated make file.
- **sctLINKKERNELDEP**. Used to implement the dependencies to recompile the kernel when it is needed.
- **sctLINKCODERLIB**. This variable should specify the `.o` files for the Coder Library source files. It will be used in the link command in the generated make file.

- **sctLINKCODERLIBDEP**. Used to implement the dependencies to re-compile the Coder Library when it is needed.

Generated Make Files

The generated make files for an SDL system will as first action include the makeoptions (make.opt) file in the directory referenced by the environment variable sctdir. It will then use the variables sctIFDEF, sctLINKKERNEL, sctCC, sctCPPFLAGS, sctCCFLAGS, sctLD, and sctLDFLAGS to compile and link the SDL system with the selected library.

The make file is generated and executed by the Cadvanced/Cbasic SDL to C Compiler.

Example 518 ----------

Below, a UNIX make file generated for the SDL system *example* is shown.

```
# makefile for System: example

sctAUTOCFGDEP =
sctCOMPFLAGS = -DXUSE_GENERIC_FUNC

include $(sctdir)/makeoptions

default: example$(sctEXTENSION)

example$(sctEXTENSION): \
  example$(sctOEXTENSION) \
  $(sctLINKKERNELDEP)
        $(sctLD) $(sctLDFLAGS) \
  example$(sctOEXTENSION) $(sctLINKKERNEL) \
  -o example$(sctEXTENSION)

example$(sctOEXTENSION): \
  example.c
        $(sctCC) $(sctCPPFLAGS) $(sctCCFLAGS) \
  $(sctIFDEF) example.c -o example$(sctOEXTENSION)
```

Adaptation to Compilers

In this section the necessary changes to the source code to adapt it to a new environment are discussed. Adapting to a new environment could mean moving the code to new hardware or using a new compiler.

There are two parts of the source code that might need changes:

1. In `scttypes.h` there is a section defining the properties of different compilers, where a new compiler can be added.
2. In `sctos.c` the functions that depend on the operating system or hardware are collected. These might need to be changed due to a new compiler, a new OS, or a new hardware.

In "Compiler Definition Section in scttypes.h" on page 3147 the compiler definition section in `scttypes.h` is discussed in detail, while `sctos.c` is treated in "The sctos.c File" on page 3149.

Compiler Definition Section in scttypes.h

Caution!

Do not to use the compiler `/usr/ucb/cc`. Our experience is that the bundled compiler is subject to generating compilation errors.

Instead, we recommend to run the unbundled compiler `/opt/SUNWSpro/bin/cc` or the GNU C compiler.

In `scttypes.h` the properties of the compiler is recognized by the compiler/computer dependent switches set by the compiler:

```
#if defined(__linux)
#define SCT_POSIX

#elif defined(__sun)
#define SCT_POSIX

#elif defined(__hpux)
#define SCT_POSIX

#elif defined(__CYGWIN__)
#define SCT_POSIX

#elif defined(QNX4_CC)
#define SCT_POSIX
```

```
#elif defined(__BORLANDC__)
#define SCT_WINDOWS

#elif defined(_MSC_VER)
#define SCT_WINDOWS

#else
#include "user_cc.h"

#endif
```

Basically this section distinguishes between Unix-like/POSIX compilers and Windows compilers. In the case the compiler is not in the list above, the user must configure it himself by writing a file `user_cc.h`, which is best placed in the target directory.

The compilers above are:

- __linux : gcc on linux
- __sun : different compilers on SUN
- __hpux : different compilers on HP
- __CYGWIN__ : gcc on windows, for more information please see http://sources.redhat.com/cygwin/
- QNX4_CC : QNX
- __BORLANDC__ : Borland compiler on Windows
- _MSC_VER : Microsoft compiler on Windows

After this compiler configuration section a general configuration section follows:

```
#if defined(SCT_POSIX) || defined(SCT_WINDOWS)
#define XMULTIBYTE_SUPPORT
#endif

#include <string.h>
#include <stdlib.h>
#include <limits.h>
#include <stdarg.h>
#ifdef XREADANDWRITEF
#include <stdio.h>
#ifdef XMULTIBYTE_SUPPORT
#include <locale.h>
#endif
#endif

#ifndef GETINTRAND
#define GETINTRAND      rand()
#endif
#ifndef GETINTRAND_MAX
#define GETINTRAND_MAX  RAND_MAX
#endif
```

```
#ifndef xptrint
#if (UINT_MAX < 4294967295)
#define xptrint         unsigned long
#define X_XPTRINT_LONG
#else
#define xptrint         unsigned
#endif
#endif

#ifndef xint32
#if (INT_MAX >= 2147483647)
#define xint32          int
#define X_XINT32_INT
#else
#define xint32          long int
#endif
#endif
```

First, the presence of multi-byte character support is set up. Then a number of standard include files are included, followed by setting up properties for random number generation. Last the two types, `xptrint`, which defines an `unsigned int` type with the same size as an address, and `xint32`, which defines a 32-bits int type, is configured.

The last three parts in this section handle the utility functions needed by `sctos.c` to implement some of the operating system dependent functions. Please see below where `sctos.c` is discussed in detail.

The sctos.c File

The following important functions are defined in `sctos.c`

```
extern void * xAlloc (xptrint Size);

extern void xFree (void **P);

extern void xHalt (void);

#ifdef XCLOCK
extern SDL_Time SDL_Clock (void);
#endif

#if defined(XCLOCK) && !defined(XENV)
extern void xSleepUntil (SDL_Time WakeUpTime);
#endif

#if defined(XPMCOMM) && !defined(XENV)
extern int xGlobalNodeNumber (void);
#endif
```

```
#if defined(XMONITOR) && !defined(XNOSELECT)
extern xbool xCheckForKeyboardInput (
  long xKeyboardTimeout);
#endif
```

Several of these functions have three different implementations, one for SCT_POSIX, one for SCT_WINDOWS and one for other cases. The other cases solution is “an empty implementation” that does not do anything. If the standard solutions in sctos.c do not fit the needs of a certain application, any of the functions above can be supplied by the user instead. By defining some of the switches:

- XUSER_ALLOC_FUNC
- XUSER_FREE_FUNC
- XUSER_HALT_FUNC
- XUSER_CLOCK_FUNC
- XUSER_SLEEP_FUNC
- XUSER_KEYBOARD_FUNC

the corresponding function or functions are removed from sctos.o and have to be supplied by the user instead.

xAlloc

The function `xAlloc` is used to allocate dynamic memory and is used throughout the runtime library and in generated code. The function is given a size in bytes and should return a pointer to a data area of the requested size. All bytes in this data area are set to zero. The standard implementation of this function uses the C function `calloc`.

A user who wants to estimate the need for dynamic memory can introduce statements in `xAlloc` to record the number of calls of `xAlloc` and the total requested size of dynamic memory. Please note two things. A program using the monitor requires more dynamic memory than a program not using the monitor, so estimates should be made with the appropriate compilation switches. A call of `calloc` will actually allocate more memory than is requested to make it possible for the C runtime system to deallocate and reuse memory. The size of this additional memory is compiler-dependent.

A user who wants to handle the case when no more memory is available at an allocation request can implement that in `xAlloc`. In the standard implementation for `xAlloc` a test if calloc returns 0 can be introduced, at which the program can be terminated with an appropriate message.

xFree

The function `xFree` is used to return memory to the list of free memory so it can be reused by subsequent calls of `xAlloc`. The standard implementation of this function uses the C function `free`. In very simple cases, no data types using dynamic memory are used and no other introduction of dynamic data by the user, this function will not be used.

The parameter of the `xFree` function, is the address of the pointer to the allocated memory.

Example 519 Using the `xFree` function ––––––––––––––––––––––––

```
unsigned char *ptr;
ptr = xAlloc(100);
xFree (&ptr);       /* NOTE: Not xFree(ptr); */
```

––

xHalt

The function `xHalt` is used to exit from a program and is in the standard implementation using the C function `exit` to perform its task.

SDL_Clock

The function `SDL_Clock` should return the current time, read from a clock somewhere in the OS or hardware. The return value is of type `SDL_Time`, that is a struct with two 32-bits integer components, representing seconds and nanoseconds in the time value.

```
typedef struct {
  xint32  s;        /* for seconds */
  xint32  ns;       /* for nanoseconds */
} SDL_Time;
```

The standard implementation of `SDL_Clock` uses the C function **`time`**, which returns the number of seconds since some defined date.

Note:

Note that the C function `time` only handles full seconds.

In an embedded system or any other application that requires better time resolution, or when the C function time is not available, `SDL_Clock` should be implemented by the user.

Note:

If an application does not require a connection with real time (for example if it is not using timers and should run as fast as possible), there is no need for a clock function. In such a case it is probably suitable to use simulated time by not defining the compilation switch `XCLOCK`, whereby `SDL_Clock` is never called and does not need to be implemented. An alternative is to let `SDL_Clock` always return the time value 0.

A typical implementation in an embedded system is to have hardware generating interrupts at a predefined rate. At each such interrupt a variable containing the current time is updated. This variable can then be read by `SDL_Clock` to return the current time.

Caution!

The variable must be protected from updates during the period of time that `the SDL_Clock` reads the clock variable.

Calling the interrupt routine while the `SDL_Clock` reads the clock variable would cause a system disaster.

xSleep_Until

The function `xSleep_Until` is given a time value, as a value of type `SDL_Time` (see above) and should suspend the executing until this time is reached, when it should return.

This function is used only when real time is used (the switch `XCLOCK` is defined) and when there is no environment functions (`XENV` is not defined). The `xSleep_Until` function is used to wait until the next event is scheduled when there is no environment that can generate events.

xGlobalNodeNumber

The function `xGlobalNodeNumber` is used to assign unique numbers to each SDL system which is part of an application.

If environment functions are used for an SDL system this function should be implemented there. If, however, we have communicating simulations, there are no `env` functions and the `xGlobalNodeNumber` function is defined in `sctos.c` instead.

So the xGlobalNodeNumber function is only used if XPMCOMM is defined and XENV is not defined. As this function is only used in the case of a communicating simulation, it is only necessary to implement it for computers/compilers that communicate with the SDL Suite, which means that it is not interesting for a user to change the standard implementation of this function. The implementation calls the function getpid, and uses thus the OS process number as global node number.

xCheckForKeyboardInput

The function xCheckForKeyboardInput is used to determine if there is a line typed on the keyboard (stdin) or not. If this is difficult to implement it can instead determine if there are any characters typed on the keyboard or not. This function is only used by the monitor system, (when XMONITOR is defined).

The xCheckForKeyboardInput function is used to implement the possibility to interrupt the execution of SDL transitions by typing <Return> and to handle polling of the environment (xInEnv or its equivalent when communicating simulations is used) when the program is waiting at the "Command :" prompt in the monitor.

List of All Compilation Switches

Introduction

This section is a reference to the macros that are used together with the generated C code from the Cadvanced/Cbasic SDL to C Compiler. Here the macros are just enumerated and explained. The section is divided in a number of subsection, each treating one major aspect of the code. Within the subsections the macros are enumerated in alphabetic order.

Information about some of the macros (Library Version Macros, Compiler Definition Section Macros, and General Properties) can also be found in the section "Compilation Switches" on page 3119.

To fully understand the descriptions of the macros in this section it is also necessary to know the basic data structures used, especially for the static structures, i.e. the xIdNodes. This information can be found in the section "The Symbol Table" on page 3028.

The information about the data types used for the dynamic structure of the system, i.e. about process instances, signal, timers, and so on, are also of interest. This can be found in "The SDL Model" on page 3070.

Library Version Macros

SCTAPPLCLENV

Application.

SCTAPPLENV

Application without clock.

SCTDEB

Stand-alone simulator for any environment. Should be executed from OS.

SCTDEBCL

Stand-alone simulator with real time for any environment. Should be executed from OS.

SCTDEBCLCOM

Simulator with real time for host. Can be executed from the SDL Suite or from OS.

SCTDEBCLENV

Stand-alone simulator, with real time and env functions, for any environment. Should be executed from OS.

SCTDEBCLENVCOM

Simulator, with real time and env functions, for any environment. May be executed from OS or from simulator GUI.

SCTDEBCOM

Simulator for host. Can be executed from the SDL Suite or from OS.

SCTOPT1APPLCLENV

Application with minimal memory requirements. Real cannot be used. No channel information

SCTOPT2APPLCLENV

Application with minimal memory requirements. Real cannot be used. Const for all channel information.

SCTPERFSIM

Suitable for execution of performance simulations.

Compiler Definition Section Macros

SCT_POSIX

Set up for UNIX/POSIX like compilers/systems.

SCT_WINDOWS

Set up for compilers on Windows

Some Configuration Macros

COMMENT(P)

Should be defined as:

```
#define COMMENT(P)
```

The macro is used to insert comments in included C code. See Example 377 on page 2683.

GETINTRAND

A random generation function. Usually `rand()` or `random()`.

GETINTRAND_MAX

The max int value generated by function mentioned in GETINTRAND. Usually RAND_MAX or 2147483647 (32-bit integers).

SCT_VERSION_6_3

Defined in generated code if the Cadvanced/Cbasic SDL to C Compiler version 6.3 was used.

XCAT(P1,P2)

Should concatenate token P1 and P2. Possibilities:

```
#define XCAT(P1,P2) P1##P2
```

or

```
#define XCAT(P1,P2) P1/**/P2
```

or

```
#define XCAT(P1,P2) XCAT2(P1)P2
#define XCAT2(P2) P2
```

XMULTIBYTE_SUPPORT

Should be set if the compiler supports multi byte characters.

XNOSELECT

Should be defined if there is no support for the select function found in UNIX operating systems. This is used to implement “user defined interrupt” by typing the return key while simulating.

XNO_VERSION_CHECK

If this macro is defined there will be no version check between the generated code and the `scttypes.h` file.

XSCT_CBASIC

Defined in generated code if Cbasic was used.

XSCT_CADVANCED

Defined in generated code if Cadvanced was used.

X_SCTTYPES_H

Defined in `scttypes.h` in a way that it possible to include the `scttypes.h` file several times without any problems.

X_XINT32_INT

Should be defined if `xint32` is `int`.

X_XPTRINT_LONG

Should be defined if `xptrint` is `unsigned long`.

General Properties

TARGETSIM

Can be used to connect an application with a monitor on a target system with the SDL Suite running on a host computer.

XASSERT

Detect and report user defined assertions that are not valid.

XCALENDARCLOCK

Use the clock function in `sctos.c` (not simulated time). Time is whatever the clock function returns.

XCLOCK

Use the clock function in `sctos.c` (not simulated time). Time is zero at system start up.

XCOVERAGE

Compile with code to store information about the current coverage of the SDL system. This information can also be printed in the monitor.

XCTRACE

Compile preserving the possibility to report the current C line number during simulations.

XEALL

Defines XEOUTPUT, XEINTDIV, XEREALDIV, XECSOP, XEFIXOF, XERANGE, XEINDEX, XECREATE, XEDECISION, XEEXPORT, XEVIEW, XEERROR, XEUNION, XECHOICE, XEOPTIONAL, XEREF, XEOWN, and XASSERT.

For more information, see these macros.

XECHOICE

Detect and report attempts to access non-active components in Choice variables.

XECREATE

Detect and report if more static instances are created at start up, than the maximum number of concurrent instances.

XECSOP

Detect and report errors in ADT operators.

XEDECISION

Detect and report when there is no possible path out from a decision.

XEERROR

Detect and report the usage of the error term in an SDL expression.

XEEXPORT

Detect and report errors in import actions.

XEFIXOF

Detect and report integer overflow in the operator fix.

XEINDEX

Detect and report index out of bounds in arrays.

XEINTDIV

Detect and report integer division with 0.

XENV

Call the env functions.

XENV_CONFORM_2_3

Insert the `VarP` pointer in the `xSignalNode` so that signals conform with their implementation in SDT 2.3.

XEOPTIONAL

Detect and report attempts to access optional struct components that are not present.

XEOUTPUT

Detect and report warnings in outputs (mainly outputs where signal is immediately discarded).

XEOWN

Detect and report illegal usage of Own and ORef pointers.

XERANGE

Detect and report subrange errors.

XEREALDIV

Detect and report real division with 0.0.

XEREF

Detect and report attempts to dereference null pointer.

XEUNION

Detect and report attempts to access non-active components in a #UNION.

XEVIEW

Detect and report errors in view actions.

XGRTRACE

Compile with the trace in source SDL graphs enabled.

XITEXCOMM

Enable the possibility for an executable to communicate with the TTCN Suite via the Postmaster.

XMAIN_NAME

If this macro is defined the main function in `sctsdl.c` will be renamed to the name given by the macro.

XMONITOR

Compile with the monitor system. This macro will implicitly set up a number of other macros as well.

XMSCE

Compile with the MSC trace enabled.

XNOMAIN

If this macro is defined the main function in `sctsdl.c` will be removed.

XPMCOMM

Enable the possibility for an executable to communicate via the Postmaster.

XPRSPRIO

Use priorities on process instance sets.

XPRSSIGPRIO

Use first priorities on process instance sets and then priorities on signal instances.

XSDLENVUI

Enable the possibility to communicate with a user-defined UI.

XSIGLOG

Call the `xSignalLog` and `xProcessLog` functions.

XSIGPRIO

Use priorities on signal instances.

XSIGPRSPRIO

Use first priorities on signal instances and then priorities on process instance sets.

XSIMULATORUI

Enable the possibility to communicate with the simulator UI.

XTENV

As XENV but call `xInEnv` at specified times (next event time is out parameter from function `xInEnv`).

XTRACE

Compile with the textual trace enabled.

Code Optimization

XCONST

The majority of the `xIdNode` structs can be made const by defining `CONST` as `const`. This is only possible in applications (not simulations).

XCONST_COMP

This should normally be defined as const if XCONST is const. It is used to introduce const in the component declarations within the `xIdNode` structs.

XNOCONTSIGFUNC

Do not include functions to calculate the expressions in continuous signals. This saves also one function pointer in the `xIdNode` for the states. If this switch is defined, continuous signals cannot be used.

XNOENABCONDFUNC

Do not include functions to calculate the expressions in enabling conditions. This saves also one function pointer in the `xIdNode` for the states. If this switch is defined, enabling conditions cannot be used.

XNOEQTIMERFUNC

Do not include function to compare the parameters of two timers. This saves also one function pointer in the `xIdNode` for the signals. If this switch is defined, timers with parameters cannot be used.

XNOREMOTEVARIDNODE

Do not include xIdNodes for remote variable definitions.

XNOSIGNALIDNODE

Do not include `xSignalIdNodes` for signals and timers.

XNOSTARTUPIDNODE

Do not include `xSignalIdNodes` for start up signals.

XNOUSEOFOBJECTIDENTIFIER

The type Object_identifier and all operations on that type are removed.

XNOUSEOFOCTETBITSTRING

The types Bit_string, Octet, Octet_string and all operations on these types are removed.

XNOUSEOFSERVICE

All data and code needed to handle services are removed.

XNOUSEOFREAL

The type real and all operations on real are removed.

XOPT

Defines XOPTSIGPARA, XOPTDCL, XOPTFPAR, XOPTSTRUCT, XOPTLIT, and XOPTSORT.

For more information see these macros.

XOPTCHAN

Do not include xIdNodes for channels, signal routes, and gates. Information in services and processes about connections to signal routes and gates are also removed.

Note:

If this compilation switch is defined all outputs must either be sent TO a process or the receiver must be possible to calculate during code generation.

XOPTDCL

Do not include xIdNodes for variables.

XOPTFPAR

Do not include xIdNodes for formal parameters.

XOPTLIT

Do not include xIdNodes for literals.

XOPTSIGPARA

Do not include xIdNodes for signal parameters.

XOPTSORT

Do not include xIdNodes for newtypes and syntypes.

XOPTSTRUCT

Do not include xIdNodes for struct components.

XPRSOPT

Optimize memory for process instances. All memory for a process instance can be reused, but signal sending to a stopped process, who's memory has been reused by a new process, cannot be detected. The new process will in this case receive the signal.

XSYNTVAR

If this compilation switch is defined, xVarIdNodes are inserted for the Present components for optional struct components. This feature is only needed by the Explorer and by LINK. It should not be defined otherwise.

Definitions of Minor Features

XBREAKBEFORE

Should be used mainly if the MONITOR or GRTRACE switches are defined. It will make the functions and struct components for SDT references available and is also used to expand the macros XAT_FIRST_SYMBOL, XBETWEEN_SYMBOLS, XBETWEEN_SYMBOLS_PRD, XBETWEEN_STMTS, XBETWEEN_STMTS_PRD, XAFTER_VALUE_RET_PRDCALL, and XAT_LAST_SYMBOL to suitable function calls. These functions are used to interrupt a transition between symbols during simulation.

XCASEAFTERPRDLABELS

See XCASELABELS below. The SDL symbols just after an SDL procedure call have to be treated specially, as the symbol number (=case label) for these symbols are used as the restart address for the calling graph. Normally this macro should be defined. If SDL procedure calls are transformed to proper C function calls, and SDL return is translated to a C return, and nextstate in a procedure is NOT translated to a C return (i.e. the process will be hanging in the C function representing the SDL procedure) then it is not necessary to define XCASEAFTERPRDLABELS.

XCASELABELS

The function implementing the behavior of a process, procedure, or service contains one large switch statement with a case label for each SDL symbol in the graph. This switch is used to be able to restart the execution of a process, procedure, or service at any symbol. In an application most of these label can be removed (all except for those symbols that start a transition, i.e. start, input, continuous signal). The macro XCASELABELS should be defined to introduce the case labels for all SDL symbol. This means that XCASELABELS should be defined in a simulation but not in an application.

XCONNECTPM

If XCONNECTPM is defined the SDL simulation will try to connect itself to the postmaster. This is necessary if GR trace (XGRTRACE), communicating simulations (XPMCOMM), or communication with the

TTCN Suite (XITEXCOMM) is to be used. The XCONNECTPM feature is normally only used in simulations.

XCOUNTRESETS

Count the number of timers that are removed at a reset operation. This information is used by the textual trace system (XTRACE) to present this information. The information is really only of interest at a stop action when more then one timer might be (implicitly) reset. XCOUNTRESETS should not be defined in an application.

XENVSIGNALLIMIT

This macro is used to determine the number of signals sent to the environment that, during simulation, should be saved in the input port of the env process instance. Such signals can be inspected with the normal monitor commands for viewing of signals. This macro is only of interest in a simulation and has the default value 20.

XERRORSTATE

Insert the data structure to represent an “error” state that can be used if no path is found out from a decision. This should normally be defined if XEDECISION is defined.

XFREESIGNALFUNCS

Insert free functions for each signal, timer, or startup signal that contains a parameter of a type having a free function. These signal free functions can the be used to free allocated data within a signal. This macro should be defined if Master Library is used.

XFREEVARS

Insert free function calls for all variables of a type with free function, just before stop or return actions. This means that free actions are performed on allocated data referred to from variables is before the object ceases to exist. This macro should be defined.

XIDNAMES

This macro is used to determine if the SDL name of an SDL object should be stored in the `xIdNode` for the object. This character string is used for communication with the user in for example the monitor. Nor-

mally this macro should not be used in an application. Sometime it might be useful for target debugging to define XIDNAMES, as it is then fairly easy to identify objects by just printing the SDL name from a debugger. On average this seems to cost approximately 5% more memory.

XNRINST

This macro should be defined if process instance numbers are to be maintained. The instance number is the number in the monitor printout `Test:2`, identifying the individual instances of the process instance set Test in this case. XNRINST is normally only used in a simulation.

XOPERRORF

Include the function `xSDLOpError` in `sctsdl.c`. This function is used to print run-time errors in ADT operators.

XPRSSENDER

Store the value of sender also in the `xPrsNode`. The normal place is in the latest received signal. This is only needed in a simulation as sender might be accessed from the monitor system after the transition is completed and the signal has been returned to the pool of available memory.

XREADANDWRITEF

Include the functions for basic Read and Write. This is needed mainly in simulations.

XREMOVETIMERSIG

Allow the removal of timer signals for not-executing PIds. This is needed only in simulations to implement the monitor commands set-timer and reset-timer.

XSIGPATH

If this macro is defined then the functions `xIsPath` and `xFindReceiver` will return the path of signal routes, channels, and gates from the sender to the receiver, as out parameters. This information can then be used in the monitor system, for example, to produce signal logs. This macro should normally not be defined in an application.

XSYMBTLINK

The XSYMBTLINK macro is used to determine if a complete tree should be built from the xIdNodes of the system. If XSYMBTLINK is defined then all xIdNodes contains a `Parent`, a `Suc`, and a `First` pointer. The value of the `Parent` pointer is generated directly into the xIdNodes. `Suc` and `First`, however, are calculated in the `yInit` function by calling the `xInsertIdNode` function. The `Suc` and `First` pointers are needed by the monitor system, but not in an application, i.e. XSYMBTLINK should be defined in a simulation but not in an application.

XTESTF

This macro is used to include or remove test functions for syntype (or newtypes) with range conditions. The `yTest` function is used by the monitor system and by the functions to test index out of bounds in arrays and to test subranges. This means that XTESTF should be defined if the monitor is used or if XERANGE or XEINDEX is defined.

XTRACHANNELSTOENV

When using partitioning of a system a problem during the redirection of channels is that the number of channels going to the environment is not known at code generation time, which means that the size of the data area used for the connections is not known. This problem is solved in two ways.

Either the function handling redirections allocates more memory, which is the default, or the user specifies how many channels that will be redirected (which could be difficult to compute, but will lead to less need of memory).

In the first case (allocation of more memory) the macros:

```
#define XTRACHANNELSTOENV  0
#define XTRACHANNELLIST
```

should be defined like above. This is the standard in scttypes.h. If the user wants to specify the number of channels himself then

```
#define XTRACHANNELSTOENV  10
#define XTRACHANNELLIST    ,0,0,0,0,0,0,0,0,0,0
```

i.e. `XTRACHANNELSTOENV` should be the number of channels, while `XTRACHANNELLIST` should be a list of that many zeros.

XTRACHANNELLIST

See `XTRACHANNELSTOENV` just above.

Static Data, Mainly xIdNodes

XBLO_EXTRAS

All generated struct values for block, block type, and block instance structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xBlockIdStruct` must be updated as well. Normally this macro should be empty.

Example 520 ----------

```
#define XBLO_EXTRAS  ,0
```

XBLS_EXTRAS

All generated struct values for block substructure structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xBlockSubstIdStruct` must be updated as well. Normally this macro should be empty.

Example 521 ----------

```
#define XBLS_EXTRAS  ,0
```

XCOMMON_EXTRAS

All generated struct values for `xIdNode` structs contain this macro after the common components. This means that it is possible to insert new components in all xIdNodes by defining this macro. Normally this macro should be empty.

Example 522 ----------

To insert a new int component with value 0 the following definition can be used:

```
#define XCOMMON_EXTRAS  ,0
```

XLIT_EXTRAS

All generated struct values for literal structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xLiteralIdStruct` must be updated as well. Normally this macro should be empty.

Example 523 ----------

```
#define XLIT_EXTRAS  ,0
```

XPAC_EXTRAS

All generated struct values for package structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xPackageIdStruct` must be updated as well. Normally this macro should be empty.

Example 524 ----------

```
#define XSYS_EXTRAS  ,0
```

XPRD_EXTRAS

All generated struct values for procedure structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xPrdIdStruct` must be updated as well. Normally this macro should be empty.

Example 525 ----------

```
#define XSYS_EXTRAS  ,0
```

XPRS_EXTRAS

`(PREFIX_PROC_NAME)`

All generated struct values for process, process type, and process instance structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xPrsIdStruct` must be updated as well.

Example 526 ----------

```
#define XPRS_EXTRAS(PREFIX_PROC_NAME) \
   ,XCAT(PREFIX_PROC_NAME,_STACKSIZE)
```

XSIG_EXTRAS

All generated struct values for signal, timer, RPC_signal, startup signal structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xSignalIdStruct` must be updated as well. Normally this macro should be empty.

Example 527 ----------

```
#define XSIG_EXTRAS  ,0
```

XSPA_EXTRAS

All generated struct values for signal parameter structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xVarIdStruct` must be updated as well (Note that variables, formal parameters, signal parameters, and struct components are all handled in `xVarIdStruct`.) Normally this macro should be empty.

Example 528 ----------

```
#define XSPA_EXTRAS  ,0
```

XSRT_EXTRAS

All generated struct values for newtype and syntype structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xSortIdStruct` must be updated as well. Normally this macro should be empty.

Example 529 ----------

```
#define XSRT_EXTRAS  ,0
```

XSRV_EXTRAS

All generated struct values for service, service type, and service instance structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xSrvIdStruct` must be updated as well. Normally this macro should be empty.

Example 530

```
#define XSRV_EXTRAS  ,0
```

XSTA_EXTRAS

All generated struct values for state structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xStateIdStruct` must be updated as well. Normally this macro should be empty.

Example 531

```
#define XSTA_EXTRAS  ,0
```

XSYS_EXTRAS

All generated struct values for system, system type, and system instance structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xSystemIdStruct` must be updated as well. Normally this macro should be empty.

Example 532

```
#define XSYS_EXTRAS  ,0
```

XSYSTEMVARS

This macro gives the possibility to introduce global variables declared in the beginning of the C file containing the implementation of the SDL system unit.

XSYSTEMVARS_H

If extern definitions are needed for the data declared in XSYSTEMVARS, this is the place to introduce it. These definitions will be present in the `.h` file for the system unit (if separate generation is used).

XVAR_EXTRAS

All generated struct values for variables, formal parameters, and struct components structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xVarIdStruct` must be updated as well (Note that signal parameters also uses the type `xVarIdStruct`). Normally this macro should be empty.

Example 533

```
#define XVAR_EXTRAS  ,0
```

Data in Processes, Procedures and Services

PROCEDURE_VARS

The struct components that are needed for each procedure instance. Example: state.

PROCESS_VARS

The struct components that are needed for each process instance. Example: state, parent, offspring, self, sender, inputport.

SERVICE_VARS

The struct components that are needed for each service instance. Example: state

YGLOBALPRD_YVARP

This macro is used to declare the `yVarP` pointer (which is a pointer to the `yVDef` struct for the process) in a procedure defined outside of a process. As a global procedure never can access process local data, it is suitable to let `yVarP` be a pointer to a struct only containing the components defined in the macro PROCESS_VARS.

YGLOBALSRV_YVARP

This macro is used to declare the `yVarP` pointer (which is a pointer to the `yVDef` struct for the process) in a service type defined outside of a process. As a global service type never can access process local data, it is suitable to let `yVarP` be a pointer to a struct only containing the components defined in the macro PROCESS_VARS.

YPAD_TEMP_VARS

Local variables in the PAD function for a process or service. Example: temporary variables needed for outputs, create actions.

YPAD_YSVARP

Declaration of the `ySVarP` pointer used to refer to the received signal. Normally `ySVarP` is `void *`.

YPAD_YVARP

(VDEF_TYPE)

This macro is used within a process and in a service defined within a process. It should be expanded to a declaration of yVarP, which is the pointer that is used to access SDL variables in the process. yVarP should be of type VDEF_TYPE *, where VDEF_TYPE is the type of the yVDef struct for the process. If the pointer to the yVDef struct is passed as parameter to the PAD function, yVarP can be assigned its correct value already in the declaration.

YPRD_TEMP_VARS

Local variables in the function implementing the behavior of an SDL procedure.

YPRD_YVARP

(VDEF_TYPE)

This macro is used within a procedure defined in a process. It should be expanded to a declaration of yVarP, which is the pointer that is used to access SDL variables in the process. yVarP should be of type VDEF_TYPE *, where VDEF_TYPE is the type of the yVDef struct for the process. If the pointer to the yVDef struct is passed as parameter to the procedure function, yVarP can be assigned its correct value already in the declaration.

Some Macro Used Within PAD Functions

BEGIN_PAD

```
(VDEF_TYPE)
```

BEGIN_PAD is a macro that can be used to insert code that is executed in the beginning of the PAD functions. `VDEF_TYPE` is the yVDef `type` for the process.

BEGIN_START_TRANSITION

```
(STARTUP_PAR_TYPE)
```

This macro can be used to introduce code that is executed at the beginning of the start transition. `STARTUP_PAR_TYPE` is the `yPDef` struct for the startup signal for this process.

CALL_SERVICE

This macro is used in the PAD function of a process that contains services. It should be expanded to a call to PAD function for the service that should execute the next transition (`ActiveSrv`).

CALL_SUPER_PAD_START

```
(PAD)
```

During the start transition of a process all inherited PAD functions up to and including the PAD function containing the START symbol have to be called. The reason is to initialize all variables defined in the process. This macro is used to perform a call to the inherited PAD function (the macro parameter PAD). Usually this macro is expanded to something like:

```
yVarP->RestartPAD = PAD; PAD(VarP);
```

followed by either a `return` or a `goto NewTransition` depending on execution model.

CALL_SUPER_PRD_START

```
(PRD, THISPRD)
```

This macro is used in the same way as CALL_SUPER_PAD_START (see above) but for the start transition in a procedure. `THISPRD` is the executing procedure function, while `PRD` is the inherited procedure function.

CALL_SUPER_SRV_START

(PAD)

This macro is used in the same way as CALL_SUPER_PAD_START (see above) but for the start transition in a service. PAD is the inherited PAD function.

LOOP_LABEL

The LOOP_LABEL macro should be used to form the loop from a next-state operation to the next input operation necessary in the OS where OS tasks does not perform return at end of transition (most commercial OS). This macro is also suitable to handle free on received signals and the treatment of the save queue. In an OS where SDL nextstate is implemented using a C `return` (the Master Library for example) the LOOP_LABEL macro is usually empty.

LOOP_LABEL_PRD

Similar to LOOP_LABEL but used in procedures with states.

LOOP_LABEL_PRD_NOSTATE

Similar to LOOP_LABEL but used in procedures without states. This macro is in many circumstances expanded to nothing.

LOOP_LABEL_SERVICEDECOMP

Similar to LOOP_LABEL but used in the PAD function for a process containing services.

SDL_OFFSPRING

Should return the value of offspring.

SDL_PARENT

Should return the value of parent.

SDL_SELF

Should return the value of self.

SDL_SENDER

Should return the value of sender.

START_SERVICES

This macro is used in the PAD function of a process that contains services. It should be expanded in such a way that the start transitions for all of the services are executed.

XEND_PRD

This is a macro generated at the end of a function that represents the behavior of a procedure. It needs not to be expanded to anything. To define it as

```
return (xbool)0;
```

might remove a compiler warning that the end of a value returning function might be reached.

XPRSNODE

Should usually be expanded to the type `xPrsNode`.

XNAMENODE

How to reach the `xPrsIdNode` from a PAD function. Normally this is `yVarP->NameNode`.

XNAMENODE_PRD

How to reach the `xPrdIdNode` from a PRD function. Normally this is `yPrdVarP->NameNode`.

XNAMENODE_SRV

How to reach the `xSrvIdNode` from a PAD function. Normally this is `ySrvVarP->NameNode`.

YPAD_FUNCTION

`(PAD)`

The function heading of the PAD function given as parameter.

YPAD_PROTOTYPE

`(PAD)`

The function prototype of the PAD function given as parameter.

YPRD_FUNCTION

```
(PRD)
```

The function heading of the PRD function given as parameter.

YPRD_PROTOTYPE

```
(PRD)
```

The function prototype of the PRD function given as parameter.

yInit Function

BEGIN_YINIT

This macro is placed in the beginning of the `yInit` function in the file containing code for the system. It can be expanded to variable declarations and initialization code.

XPROCESSDEF_C

```
(PROC_NAME, PROC_NAME_STRING, PREFIX_PROC_NAME,
PAD_FUNCTION, VDEF_TYPE)
```

This macro can be used to introduce code for each process instance set in the system.

Parameters:

- `PROC_NAME`
  the name of the process without prefix.
- `PROC_NAME_STRING`
  the name of the process as a character string.
- `PREFIX_PROC_NAME`
  the name of the process with prefix.
- `PAD_FUNCTION`
  the PAD function for this process instance set.
- `VDEF_TYPE`
  the `yVDef` struct for this process.

XPROCESSDEF_H

```
(PROC_NAME, PROC_NAME_STRING, PREFIX_PROC_NAME,
PAD_FUNCTION, VDEF_TYPE)
```

This macro can be used to introduce extern declaration (placed in the proper `.h` file) for each process instance set in the system.

Parameters:

- `PROC_NAME`
  the name of the process without prefix.
- `PROC_NAME_STRING`
  the name of the process as a character string.
- `PREFIX_PROC_NAME`
  the name of the process with prefix.
- `PAD_FUNCTION`
  the PAD function for this process instance set.
- `VDEF_TYPE`
  the `yVDef` struct for this process.

xInsertIdNode

In the `yInit` function the function `xInsertIdNode` is called for each `IdNode`. In an application this is not necessary, and `xInsertIdNode` can be defined as

```
#define xInsertIdNode(Node)
```

The function `xInsertIdNode` is needed if XSYMBTLINK, XCOVERAGE, or XMONITOR is defined.

YINIT_TEMP_VARS

This macro is placed in all yInit functions and can be expanded to local variables needed within the `yInit` function.

Implementation of Signals and Output

ALLOC_SIGNAL

ALLOC_SIGNAL_PAR

```
(SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_TYPE)
```

These macros are used to allocate a data area for a signal to be sent. ALLOC_SIGNAL is used if the signal has no parameters, while ALLOC_SIGNAL_PAR is used if the signal has parameters. The resulting data area should be reference by the variable mentioned by the macro OUTSIGNAL_DATA_PTR (see below).

Parameters:

- `SIG_NAME`
  the name of the signal without prefix.
- `SIG_IDNODE`
  the `xSignalIdNode` of the signal.
- `RECEIVER`
  the receiver given in the TO clause, or calculated. In a NO_TO output, `RECEIVER` is `xNotDefPId`.
- `SIG_PAR_TYPE`
  the yPDef `type` of the signal. If the signal has no parameters this macro parameter is XSIGNALHEADERTYPE (see below).

INSIGNAL_NAME

This macro should be expanded to the identification of the currently received signal. It is used to distinguish between signals when several signal is enumerated in the same input symbol.

OUTSIGNAL_DATA_PTR

This should be the pointer referring to be signal data area while building the signal during an output. It should be assigned its value in ALLOC_SIGNAL or ALLOC_SIGNAL_PAR, and will then be used during assignment of signal parameters and in the SDL_2OUTPUT macro just below.

SDL_2OUTPUT

SDL_2OUTPUT_NO_TO

SDL_2OUTPUT_COMPUTED_TO

SDL_ALT2OUTPUT

SDL_ALT2OUTPUT_NO_TO

SDL_ALT2OUTPUT_COMPUTED_TO

```
(PRIO, VIA, SIG_NAME, SIG_IDNODE, RECEIVER,
SIG_PAR_SIZE, SIG_NAME_STRING)
```

These six macros are used to send the signal created in ALLOC_SIGNAL or ALLOC_SIGNAL_PAR. The SDL_ALT versions of the macros are used if the directive `/*#ALT*/` has been given in the output. The version without suffix is used for an output TO, while the suffix _COMPUTED_TO is used for an output without to but it was possible to compute the receiver during code generation time. The suffix _NO_TO indicates an output without to, where the receiver cannot be calculated during code generation time.

Parameters:

- `PRIO`
  the priority of the signal specified in a #PRIO directive.
- `VIA`
  the via list given in the output.
- `SIG_NAME`
  the name of the signal without prefix
- `SIG_IDNODE`
  the `xSignalIdNode` for the signal.
- `RECEIVER`
  the receiver given in the TO clause, or calculated. In a NO_TO output, `RECEIVER` is `xNotDefPId`.
- `SIG_PAR_SIZE`
  the size of the `yPDef` struct of the signal. If signal without parameters `SIG_PAR_SIZE` is 0.
- `SIG_NAME_STRING`
  the name of the signal as a character string.

SDL_THIS

In an output TO THIS in SDL, the RECEIVER parameter in the ALLOC_SIGNAL and SDL_2OUTPUT macros discussed above will become SDL_THIS.

SIGCODE

```
(P)
```

This macro makes it possible to store a signal code (signal number) in the `xSignalIdNode` for a signal. The macro parameter `P` is the signal name without prefix.

SIGNAL_ALLOC_ERROR

This macro is inserted after the ALLOC_SIGNAL macro and the assignment of parameter values to the signal. It can be used to test if the alloc was successful or not.

SIGNAL_ALLOC_ERROR_END

This macro is inserted after the SDL_2OUTPUT macro.

SIGNAL_NAME

```
(SIG_NAME, SIG_IDNODE)
```

This macro should be expanded to an identification of the signal given as parameter. Normally the identification is either the `xSignalIdNode` for the signal or an `int` value. If the id is an `int` value it is suitable to insert defines of type #define signal_name number. A file containing such defines can be generated using the Generate Signal Numbers feature in Cadvanced/Cbasic.

Parameters:

- `SIG_NAME`
  the name of the signal without parameters
- `SIG_IDNODE`
  the `xSignalIdNode` for the signal.

SIGNAL_VARS

The struct components that are needed for each signal instance. Example: sender, receiver, signal type.

TO_PROCESS

```
(PROC_NAME, PROC_IDNODE)
```

This macro is used as RECEIVER in the ALLOC_SIGNAL and SDL_2OUTPUT macros if the signal is sent to a process instance set in SDL.

Parameters:

- `PROC_NAME`
  the name of the receiving process without prefix.
- `PROC_IDNODE`
  the `xPrsIdNode` of the receiving process.

TRANSFER_SIGNAL

TRANSFER_SIGNAL_PAR

```
(SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_TYPE)
```

These macros are used as alternative for the ALLOC_SIGNAL macros (see these macros above) if the directive `#TRANSFER` if given in the output.

- `SIG_NAME`
  the name of the signal without prefix.
- `SIG_IDNODE`
  the `xSignalIdNode` of the signal.
- `RECEIVER`
  the receiver given in the TO clause, or calculated. In a NO_TO output, `RECEIVER` is `xNotDefPId`.
- `SIG_PAR_TYPE`
  the `yPDef` type of the signal. If the signal has no parameters this macro parameter is XSIGNALHEADERTYPE (see below).

XNONE_SIGNAL

The representation for a none signal.

XSIGNALHEADERTYPE

This macro is used to indicate a `yPDef` struct for a signal without parameters. Such a signal has no generated `yPDef` struct. It is suitable to let

XSIGNALHEADERTYPE be the name of a struct just containing the components in SIGNAL_VARS.

XSIGTYPE

Depending on the representation of the signal type that is used (`xSignalIdNode` or `int`) this macro should either be `xSignalIdNode` or `int`.

Implementation of RPC

ALLOC_REPLY_SIGNAL

ALLOC_REPLY_SIGNAL_PAR

ALLOC_REPLY_SIGNAL_PRD

ALLOC_REPLY_SIGNAL_PRD_PAR

```
(SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_TYPE)
```

These macros are used to allocate the Reply signal in the signal exchange in an RPC. The suffix _PAR is used if the reply signal contains parameters. The suffix _PRD is used if the implicit RPC transition is part of a procedure.

Parameters:

- `SIG_NAME`
  the reply signal name without prefix.
- `SIG_IDNODE`
  the `xSignalIdNode` for the reply signal.
- `RECEIVER`
  the receiver of the reply signal. The macro XRPC_SENDER_IN_ALLOC or XRPC_SENDER_IN_ALLOC_PRD are used as actual parameter. The suffix _PRD is used if the implicit RPC transition is part of a procedure.
- `SIG_PAR_TYPE`
  the `yPDef` type for the reply signal. If the reply signal does not contain any parameters the macro name XSIGNALHEADERTYPE is generated as actual parameter.

REPLYSIGNAL_DATA_PTR

REPLYSIGNAL_DATA_PTR_PRD

This should be a reference to the data area for the reply signal that is allocated in the ALLOC_REPLY_SIGNAL macro. The suffix _PRD is used if the implicit RPC transition is part of a procedure.

SDL_RPCWAIT_NEXTSTATE

SDL_RPCWAIT_NEXTSTATE_PRD

```
(PREPLY_IDNODE, PREPLY_NAME, RESTARTADDR)
```

These macros are used to implement the implicit nextstate operation in the caller of an RPC. The suffix _PRD is used if the implicit RPC transition is part of a procedure.

Parameters:

- `PREPLY_IDNODE`
  the `xSignalIdNode` for the reply signal.
- `PREPLY_NAME`
  the name without prefix for the reply signal.
- `RESTARTADDR`
  the restart address (symbol number) for the implicit input of the reply signal.

SDL_2OUTPUT_RPC_CALL

```
(PRIO, VIA, SIG_NAME, SIG_IDNODE, RECEIVER,
SIG_PAR_SIZE, SIG_NAME_STRING)
```

Send the call signal of an RPC.

Parameters:

- `PRIO`
  priority of signal.
- `VIA`
  the via list, which in this case always is `(xIdNode *)0`, i.e. no via list.
- `SIG_NAME`
  the RPC call signal name without prefix.
- `SIG_IDNODE`
  the `xSignalIdNode` for the RPC call signal.

- RECEIVER
  the receiver of the call signal. This is either expressed as an ordinary TO-expression or using the macro XGETEXPORTINGPRS (see below) in case of no explicit receiver specified in the call.

- SIG_PAR_SIZE
  the size of the yPDef struct for the call signal. If the call signal has no parameters this parameter will be 0.

- SIG_NAME_STRING
  the name of the RPC call signal as a character string.

SDL_2OUTPUT_RPC_REPLY

SDL_2OUTPUT_RPC_REPLY_PRD

```
(PRIO, VIA, SIG_NAME, SIG_IDNODE, RECEIVER,
SIG_PAR_SIZE, SIG_NAME_STRING)
```

These macros are used to send the RPC reply signal. The suffix _PRD is used if the implicit RPC transition is part of a procedure.

Parameters:

- PRIO
  priority of signal.

- VIA
  the via list, which in this case always is `(xIdNode *)0`, i.e. no via list.

- SIG_NAME
  the RPC reply signal name without prefix.

- SIG_IDNODE
  the `xSignalIdNode` for the RPC reply signal.

- RECEIVER
  the receiver of the reply signal. This is expressed using the macro XRPC_SENDER_IN_OUTPUT or XRPC_SENDER_IN_OUTPUT_PRD.

- SIG_PAR_SIZE
  the size of the yPDef struct for the reply signal. If the reply signal has no parameters this parameter will be 0.

- SIG_NAME_STRING
  the name of the RPC reply signal as a character string.

XGETEXPORTINGPRS

```
(REMOTENODE)
```

This macro should be expanded to an expression that given the remote procedure given as actual macro parameter (more exactly the `IdNode` for the remote procedure), returns one possible exporter of this remote procedure. Usually this macro is expanded to a call of the library function `xGetExportingPrs`.

XRPC_REPLY_INPUT

XRPC_REPLY_INPUT_PRD

Macros that can be used for special processing needed to receive an RPC reply signal. The macros are usually expanded to nothing.

XRPC_SAVE_SENDER

XRPC_SAVE_SENDER_PRD

These macros can be used to save the sender of a received RPC call signal, for further use when the reply signal is to be sent. The suffix _PRD is used if the implicit RPC transition is part of a procedure.

XRPC_SENDER_IN_ALLOC

XRPC_SENDER_IN_ALLOC_PRD

These macros are used to obtain the receiver of the reply signal (from the sender of the call signal) in the ALLOC_REPLY_SIGNAL macros. The suffix _PRD is used if the implicit RPC transition is part of a procedure.

XRPC_SENDER_IN_OUTPUT

XRPC_SENDER_IN_OUTPUT_PRD

These macros are used to obtain the receiver of the reply signal (from the sender of the call signal) in the SDL_2OUTPUT_RPC_REPLY macros. The suffix _PRD is used if the implicit RPC transition is part of a procedure.

XRPC_WAIT_STATE

The state number used for a RPC wait state. XRPC_WAIT_STATE is usually defined as -3.

Implementation of View and Import

XGETEXPORTADDR

```
(REMOTENODE, EXPORTER, IS_DEF_EXPORTER)
```

This macro should be expanded to an expression that returns the address of the exported variable. Usually the function `xGetExportAddr` is called.

Parameters:

- `REMOTENODE`
  the IdNode for the remote variable.
- `EXPORTER`
  the value of the optional PId expression. If no PId expression is given this parameter is SDL_NULL.
- `IS_DEF_EXPORTER`
  has the value `(xbool)1` if a PId expression was found in the import statement, otherwise it is `(xbool)0`.

SDL_VIEW

```
(PID_EXPR, HAS_EXPR, VAR_NAME_STRING, REVEALED_LIST,
SORT_SIZE)
```

This macro should be expanded to an expression that returns the address of the viewed variable. Usually the function SDL_View is called.

Parameters:

- `PID_EXPR`
  the value of the optional PId expression. If no PId expression is given this parameter is SDL_NULL.
- `HAS_EXPR`
  has the value `(xbool)1` if a PId expression was found in the view statement, otherwise it is `(xbool)0`.
- `VAR_NAME_STRING`
  the name of the viewed variable as a character string.
- `REVEALED_LIST`
  the list of the revealed variables.
- `SORT_SIZE`
  the size of the sort of the revealed variable.

Implementation of Static and Dynamic Create and Stop

ALLOC_STARTUP

ALLOC_STARTUP_PAR

```
(PROC_NAME, STARTUP_IDNODE, STARTUP_PAR_TYPE)
```

Allocate the data area for a startup signal and let the pointer mentioned in the macro STARTUP_DATA_PTR refer to this data area. The suffix _PAR is used if the startup signal contains parameters.

Parameters:

- `PROC_NAME`
  the name without prefix for the created process.
- `STARTUP_IDNODE`
  the `xSignalIdNode` for the startup signal of the created process.
- `STARTUP_PAR_TYPE`
  the `yPDef` for the startup signal of the created process.

ALLOC_STARTUP_THIS

Allocate the data area for a startup signal and let the pointer mentioned in the macro STARTUP_DATA_PTR refer to this data area. This macro is used in a create THIS operation.

INIT_PROCESS_TYPE

```
(PROC_NAME, PREFIX_PROC_NAME, PROC_IDNODE,
PROC_NAME_STRING, MAX_NO_OF_INST, STATIC_INST,
VDEF_TYPE, PRIO, PAD_FUNCTION)
```

This macro will be call once for each process instance set in the yInit function. It should be used to initiated common features for all instances of a process instance set.

Parameters:

- `PROC_NAME`
  the name without prefix for the process instance set.
- `PREFIX_PROC_NAME`
  the name with prefix for the process instance set.
- `PROC_IDNODE`
  the `xPrsIdNode` for the process instance set.

- PROC_NAME_STRING
  the name as character string for the process instance set.
- MAX_NO_OF_INST
  the maximum number of instances of this process instance set.
- STATIC_INST
  the number of static instances of this process instance set.
- VDEF_TYPE
  the yVDef type for this process instance set.
- PRIO
  the priority for process instance set.
- PAD_FUNCTION
  the PAD for this process instance set.

SDL_CREATE

```
(PROC_NAME, PROC_IDNODE, PROC_NAME_STRING)
```

This macro is used to create (a create action) a process instance.

Parameters:

- PROC_NAME
  the name without prefix for the process instance set.
- PROC_IDNODE
  the xPrsIdNode for the process instance set.
- PROC_NAME_STRING
  he name as character string for the process instance set.

SDL_CREATE_THIS

This macro is used to implement create this.

SDL_STATIC_CREATE

```
(PROC_NAME, PREFIX_PROC_NAME, PROC_IDNODE,
PROC_NAME_STRING, STARTUP_IDNODE, STARTUP_PAR_TYPE,
VDEF_TYPE, PRIO, PAD_FUNCTION, BLOCK_INST_NUMBER)
```

This macro is called in the yInit function once for each static process instances that should be created of a process instance set.

Parameters:

- `PROC_NAME`
  the name without prefix for the process instance set.
- `PREFIX_PROC_NAME`
  the name with prefix for the process instance set.
- `PROC_IDNODE`
  the `xPrsIdNode` for the process instance set.
- `PROC_NAME_STRING`
  the name as character string for the process instance set.
- `STARTUP_IDNODE`
  the `xSignalIdNode` for the startup signal for the process instance set.
- `STARTUP_PAR_TYPE`
  the `yPDef` type for the startup signal for the process instance set.
- `VDEF_TYPE`
  the `yVDef` type for the process instance set.
- `PRIO`
  the priority for the process instance set.
- `PAD_FUNCTION`
  the PAD function for the process instance set.
- `BLOCK_INST_NUMBER`
  if this process instance set is part if a block instance set then this macro is the block instance number for the block instance set that this process belongs to. Otherwise this macro parameter is 1.

SDL_STOP

This macro is used to implement the SDL operation stop (both in processes and in services).

STARTUP_ALLOC_ERROR

This macro is inserted after the ALLOC_STARTUP macro and the assignment of parameter values to the signal. It can be used to test if the alloc was successful or not.

STARTUP_ALLOC_ERROR_END

This macro is inserted after the SDL_CREATE macro.

STARTUP_DATA_PTR

This macro should be expanded to a temporary variable used to store a reference to the startup signal data area. It should be assigned in the ALLOC_STARTUP macro and will be used to assign the actual signal parameters (the fpar values) to the startup signal.

STARTUP_VARS

This macro can be used to insert additional general components in the startup signals. In all startup signal yPDef structs SIGNAL_VARS will be followed by STARTUP_VARS.

Implementation of Timers, Timer Operations and Now

ALLOC_TIMER_SIGNAL_PAR

```
(TIMER_NAME, TIMER_IDNODE, TIMER_PAR_TYPE)
```

Allocate a data area for the timer signal with parameters.

Parameters:

- `TIMER_NAME`
  the name without prefix of the timer.
- `TIMER_IDNODE`
  the `xSignalIdNode` for the timer.
- `TIMER_PAR_TYPE`
  the `yPDef` for the timer.

DEF_TIMER_VAR

DEF_TIMER_VAR_PARA

```
(TIMER_VAR)
```

There will be one application of this macro in the `yVDef` type for the process for each timer declaration the process contains. These declarations can be used to introduce components (timer variables) in the `yVDef` struct to track timers. The parameter `TIMER_VAR` is a suitable name for such a variable. The suffix _PARA is used if the timer has parameters.

INIT_TIMER_VAR

INIT_TIMER_VAR_PARA

```
(TIMER_VAR)
```

These macros will be inserted in start transitions, during initialization of process variables. This makes it possible to initialize the timer variables that might be inserted in the DEF_TIMER_VAR macro. The parameter `TIMER_VAR` is the name for such a variable. The suffix _PARA is used if the timer has parameters.

INPUT_TIMER_VAR

INPUT_TIMER_VAR_PARA

```
(TIMER_VAR)
```

These macros will be inserted at an input operation on a timer signal. This makes it possible to update the timer variables that might be inserted in the DEF_TIMER_VAR macro. The parameter `TIMER_VAR` is the name for such a variable. The suffix _PARA is used if the timer has parameters. Note that if a timer signal is received in an input * statement, no INPUT_TIMER_VAR will be present in this case.

RELEASE_TIMER_VAR

RELEASE_TIMER_VAR_PARA

```
(TIMER_VAR)
```

These macros will be inserted at a stop. This makes it possible to perform cleaning up of the timer variables that might be inserted in the DEF_TIMER_VAR macro. The parameter `TIMER_VAR` is the name for such a variable. The suffix _PARA is used if the timer has parameters.

SDL_ACTIVE

```
(TIMER_NAME, TIMER_IDNODE, TIMER_VAR)
```

This macro is used to implement the SDL operation active on a timer. Note that active on timers with parameters is not implemented in the Cadvanced/Cbasic SDL to C Compiler.

Parameters:

- `TIMER_NAME`
  the name without prefix of the timer.
- `TIMER_IDNODE`
  the xSignalIdNode for the timer.
- `TIMER_VAR`
  the timer variable that might be inserted in the macro DEF_TIMER_VAR.

SDL_NOW

This is the implementation of now in SDL.

SDL_RESET

```
(TIMER_NAME, TIMER_IDNODE, TIMER_VAR,
TIMER_NAME_STRING)
```

This macro is used to implement the SDL operation reset on a timer without parameters.

Parameters:

- `TIMER_NAME`
  the name without prefix of the timer.
- `TIMER_IDNODE`
  the `xSignalIdNode` for the timer.
- `TIMER_VAR`
  the timer variable that might be inserted in the macro DEF_TIMER_VAR.
- `TIMER_NAME_STRING`
  the name of the timer as a character string.

SDL_RESET_WITH_PARA

```
(EQ_FUNC, TIMER_VAR, TIMER_NAME_STRING)
```

This macro is used to implement the SDL operation reset on a timer with parameters. Before this macro a timer signal with the timer parameters in the reset operation is created.

Parameters:

- EQ_FUNC
  the name of the generated equal function that can test if two timer instance are equal or not.

- TIMER_VAR
  the timer variable that might be inserted in the macro DEF_TIMER_VAR.

- TIMER_NAME_STRING
  the name of the timer as a character string.

SDL_SET

```
(TIME_EXPR, TIMER_NAME, TIMER_IDNODE, TIMER_VAR,
TIMER_NAME_STRING)
```

SDL_SET_WITH_PARA

```
(TIME_EXPR, TIMER_NAME, TIMER_IDNODE,
TIMER_PAR_TYPE, EQ_FUNC, TIMER_VAR,
TIMER_NAME_STRING)
```

SDL_SET_DUR

```
(TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE,
TIMER_VAR, TIMER_NAME_STRING)
```

SDL_SET_DUR_WITH_PARA

```
(TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE,
TIMER_PAR_TYPE, EQ_FUNC, TIMER_VAR,
TIMER_NAME_STRING)
```

SDL_SET_TICKS

```
(TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE,
TIMER_VAR, TIMER_NAME_STRING)
```

SDL_SET_TICKS_WITH_PARA

```
(TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE,
TIMER_PAR_TYPE, EQ_FUNC, TIMER_VAR,
TIMER_NAME_STRING)
```

These six SDL_SET macros are used to implement the SDL operation set on a timer. The suffix _WITH_PARA indicates the set of a timer with parameters. In this case the SDL_SET macro is preceded by an ALLOC_TIMER_SIGNAL_PAR macro call, plus the assignment of the timer parameters. The suffix _DUR is used if the time value in the set operation is expressed as:

```
now + expression
```

In this case both the time value and the duration value (the expression above) is available as macro parameter. The suffix _TICKS is used if the time value in the set operation is expressed as:

```
now + TICKS(...)
```

where TICKS is an operator returning a duration value. In this case both the time value and the duration value (the TICKS expression above) is available as macro parameter.

Parameters:

- `TIME_EXPR`
  the time expression.
- `DUR_EXPR`
  the duration expression (only in _DUR and _TICKS).
- `TIMER_NAME`
  the timer name without prefix.
- `TIMER_IDNODE`
  the `xSignalIdNode` for the timer.
- `TIMER_PAR_TYPE`
  the `yPDef` struct for the timer (only in _WITH_PARA)
- `EQ_FUNC`
  the function that can be used to test if two timers have the same parameter values (only in _WITH_PARA).
- `TIMER_VAR`
  the name of the timer variable that might be introduced in the macro DEF_TIMER_VAR.
- `TIMER_NAME_STRING`
  the name of the timer as a character string.

TIMER_DATA_PTR

This should be the pointer referring to be timer data area while building the timer. It should be assigned its value in ALLOC_TIMER_SIGNAL_PAR, and will then be used during assignment of signal parameters and in the SDL_SET macro

TIMER_SIGNAL_ALLOC_ERROR

This macro is inserted after the ALLOC_TIMER_SIGNAL_PAR macro and the assignment of parameter values to the timer. It can be used to test if the alloc was successful or not.

TIMER_SIGNAL_ALLOC_ERROR_END

This macro is inserted after the SDL_SET macro.

TIMER_VARS

The struct components that are needed for each timer instance. Example: sender, receiver, timer type.

As timers are signals as well, after the timer signal has been sent, TIMER_VARS has to be identical to SIGNAL_VARS, except that new component may be add last in TIMER_VARS.

XTIMERHEADERTYPE

This macro is used to indicate a `yPDef` struct for a timer without parameters. Such a timer has no generated `yPDef` struct. It is suitable to let XTIMERHEADERTYPE be the name of a struct just containing the components in TIMER_VARS.

Implementation of Call and Return

ALLOC_PROCEDURE

```
(PROC_NAME, PROC_IDNODE, VAR_SIZE)
```

Allocate a data area (`yVDef`) for the called procedure.

Parameters:

- `PROC_NAME`
  the name of procedure with prefix.
- `PROC_IDNODE`
  the `xPrdIdNode` of the called procedure
- `VAR_SIZE`
  the size of the `yVDef` struct for the procedure.

ALLOC_THIS_PROCEDURE

Allocate a data area (`yVDef`) for a procedure when call THIS is used.

ALLOC_VIRT_PROCEDURE

```
(PROC_IDNODE)
```

Allocate a data area (`yVDef`) for the called procedure when calling a virtual procedure. The `PROC_IDNODE` parameter is the `xPrdIdNode` for the call procedure.

CALL_PROCEDURE

CALL_PROCEDURE_IN_PRD

```
(PROC_NAME, PROC_IDNODE, LEVELS, RESTARTADDR)
```

These macros are used to implement a call operation in SDL. The `yVDef` struct has been allocated earlier (in ALLOC_PROCEDURE) and the actual parameters have been assigned to components in this struct. The suffix _IN_PRD indicates that the procedure call is made in a procedure.

Parameters:

- `PROC_NAME`
  the name of procedure with prefix, which is the same as the name of the C function representing the behavior of the procedure.
- `PROC_IDNODE`
  the `xPrdIdNode` of the called procedure.
- `LEVELS`
  the scope level between the caller and the called procedure.
- `RESTARTADDR`
  the restart address the symbol number for the symbol after the procedure call.

CALL_PROCEDURE_STARTUP

CALL_PROCEDURE_STARTUP_SRV

These two macros are only of interest if the PAD functions are left via a return at the end of transitions. In that case any outstanding procedure must be restarted when the process becomes active again.

CALL_THIS_PROCEDURE

```
(RESTARTADDR)
```

This macro is used to implement a call THIS operation in SDL. `RESTARTADDR` is the restart address the symbol number for the symbol after the procedure call.

CALL_VIRT_PROCEDURE

CALL_VIRT_PROCEDURE_IN_PRD

```
(PROC_IDNODE, LEVELS, RESTARTADDR)
```

These macros are used to implement a call operation on a virtual procedure in SDL. The `yVDef` struct has been allocated earlier (in ALLOC_VIRT_PROCEDURE) and the actual parameters have been assigned to components in this struct. The suffix _IN_PRD indicates that the procedure call is made in a procedure.

Parameters:

- `PROC_IDNODE`
  the xPrdIdNode of the called procedure.
- `LEVELS`
  the scope level between the caller and the called procedure.
- `RESTARTADDR`
  the restart address the symbol number for the symbol after the procedure call.

PROCEDURE_ALLOC_ERROR

This macro is inserted after the ALLOC_PROCEDURE macro and the assignment of parameter values to the procedure parameters. It can be used to test if the alloc was successful or not.

PROCEDURE_ALLOC_ERROR_END

This macro is inserted after the CALL_PROCEDURE macro.

PROC_DATA_PTR

This macro should be expanded to a temporary variable used to store a reference to the procedure data area. It should be assigned in the ALLOC_PROCEDURE macro and will be used to assign the actual procedure parameters (the fpar values).

SDL_RETURN

The implementation of return in SDL.

XNOPROCATSTARTUP

If this macro is defined then all the code discussed for the macro CALL_PROCEDURE_STARTUP (just above) is removed.

Implementation of Join

Joins in SDL are normally implemented as goto:s in C, but in one case a more complex implementation is needed. This is when the label, mentioned in the join, is in a super type.

XJOIN_SUPER_PRS

`(RESTARTADDR,RESTARTPAD)`

XJOIN_SUPER_PRD

`(RESTARTADDR,RESTARTPRD)`

XJOIN_SUPER_SRV

`(RESTARTADDR,RESTARTSRV)`

These macros represent join to super type in processes, procedures, and services, in that order.

Parameters:

- `RESTARTADDR`
  The restart address in the super type.
- `RESTARTPAD, RESTARTPRD, RESTARTSRV`
  The PAD function for the super type.

Implementation of State and Nextstate

> **Note:**
>
> Implicit nextstate operations in RPC calls are treated in the RPC section.

ASTERISK_STATE

The state number for an asterisk state. ASTERISK_STATE is usually defined as -1.

ERROR_STATE

The state number used for the error state. ERROR_STATE is usually defined as -2.

START_STATE

The state number for the start state. START_STATE should be defined as 0.

START_STATE_PRD

The state number for the start state in a procedure. START_STATE_PRD should be defined as 0.

SDL_NEXTSTATE

```
(STATE_NAME, PREFIX_STATE_NAME, STATE_NAME_STRING)
```

Nextstate operation (in process or service) of the given state.

Parameters:

- `STATE_NAME`
  the name without prefix of the state.
- `PREFIX_STATE_NAME`
  the name with prefix for the state. This identifier is defined as a suitable state number in generated code and is usually used as the representation of the state.
- `STATE_NAME_STRING`
  the name of the state as a character string.

SDL_DASH_NEXTSTATE

Dash nextstate operation in a process.

SDL_DASH_NEXTSTATE_SRV

Dash nextstate operation in a service.

SDL_NEXTSTATE_PRD

```
(STATE_NAME, PREFIX_STATE_NAME, STATE_NAME_STRING)
```

Nextstate operation (in procedure) of the given state.

Parameters:

- STATE_NAME
  the representation of the state.
- PREFIX_STATE_NAME
  the name with prefix for the state. This identifier is defined as a suitable state number in generated code and is usually used as the representation of the state.
- STATE_NAME_STRING
  the name of the state as a character string.

SDL_DASH_NEXTSTATE_PRD

Dash nextstate operation in a procedure.

Implementation of Any Decisions

An any decision with two paths are generated according to the following structure:

```
BEGIN_ANY_DECISION(2)
DEF_ANY_PATH(1, 2)
DEF_ANY_PATH(2, 0)
END_DEFS_ANY_PATH(2)
BEGIN_FIRST_ANY_PATH(1)
  statements
END_ANY_PATH
BEGIN_ANY_PATH(2)
  statements
END_ANY_PATH
END_ANY_DECISION
```

BEGIN_ANY_DECISION

```
(NO_OF_PATHS)
```

Start of the any decision. NO_OF_PATHS is the number of paths in the decision.

BEGIN_ANY_PATH

```
(PATH_NO)
```

A path (not the first) in implementation part of the any decision. PATH_NO is the path number.

BEGIN_FIRST_ANY_PATH

```
(PATH_NO)
```

The first possible path in implementation part of the any decision. PATH_NO is the path number.

DEF_ANY_PATH

(PATH_NO, SYMBOLNUMBER)

Definition of a path in the decision.

Parameters:

- PATH_NO
  the path number.
- SYMBOLNUMBER
  the symbol number for the first symbol in this path.

END_ANY_DECISION

The end of the any decision.

END_ANY_PATH

End of one of the paths in the implementation section.

END_DEFS_ANY_PATH

(NO_OF_PATHS)

End of the definition part of the any decision. NO_OF_PATHS is the number of paths in the decision.

Implementation of Informal Decisions

The implementation of informal decisions are similar to any decisions.

BEGIN_FIRST_INFORMAL_PATH

(PATH_NO)

The first possible path in implementation part of the informal decision. PATH_NO is the path number.

BEGIN_INFORMAL_DECISION

(NO_OF_PATHS, QUESTION)

Start of the any decision.

Parameters:

- NO_OF_PATHS
  the number of paths in the decision.
- QUESTION
  the question charstring.

BEGIN_INFORMAL_ELSE_PATH

(PATH_NO)

The else path in implementation part of the any decision. PATH_NO is the path number.

BEGIN_INFORMAL_PATH

(PATH_NO)

A path in implementation part of the any decision. PATH_NO is the path number.

DEF_INFORMAL_PATH

(PATH_NO, ANSWER, SYMBOLNUMBER)

Definition of a path in the decision.

Parameters:

- PATH_NO
  the path number.
- ANSWER
  the answer string.
- SYMBOLNUMBER
  the symbol number for the first symbol in this path.

DEF_INFORMAL_ELSE_PATH

(PATH_NO, SYMBOLNUMBER)

Definition of the else path in the decision.

Parameters:

- PATH_NO
  the path number.
- SYMBOLNUMBER
  the symbol number for the first symbol in this path.

END_DEFS_INFORMAL_PATH

```
(NO_OF_PATHS)
```

End of the definition part of the informal decision. `NO_OF_PATHS` is the number of paths in the decision.

END_INFORMAL_ELSE_PATH

End of the else paths in the implementation section.

END_INFORMAL_DECISION

The end of the informal decision.

END_INFORMAL_PATH

End of one of the paths in the implementation section.

Macros for Component Selection Tests

The macros in this section handles testing the validity of for example a component selection of a choice or #UNION variable. Also tests for optional components in structs and for de-referencing of pointers is treated here.

XCHECK_CHOICE_USAGE

```
(TAG,VALUE,NEQTAG,COMPNAME,CURR_VALUE,TYPEINFO)
```

XSET_CHOICE_TAG

```
(TAG,VALUE,ASSTAG,NEQTAG,COMPNAME,CURR_VALUE,
 TYPEINFO)
```

XSET_CHOICE_TAG_FREE

```
(TAG,VALUE,ASSTAG,NEQTAG,FREEFUNC,COMPNAME,
 CURR_VALUE,TYPEINFO)
```

The CHOICE macros are used to test and to set the implicit tag in a choice variable. The XSET_CHOICE_TAG and XSET_CHOICE_TAG_FREE set the tag when some component of the choice is assigned a value. The FREE version of the macro is used if the choice contains some component that has a Free function. The XCHECK_CHOICE_USAGE is used to test if an accessed component is active or not.

Parameters:

- TAG
  The implicit tag component
- VALUE
  The new or expected tag value
- ASSTAG
  The assignment function for the tag type
- NEQTAG
  The equal test function for the tag type
- FREEFUNC
  The Free function for the Choice type
- COMPNAME
  The name of the selected component as a char string
- CURR_VALUE
  The current value of the tag type
- TYPEINFO
  The type info node for the tag type.

XCHECK_OPTIONAL_USAGE

```
(PRESENT_VAR,COMPNAME)
```

This macro is used to check that a selected optional component is present. The PRESENT_VAR parameter is the present variable for this component, while COMPNAME is the selected components name as a char string.

XCHECK_UNION_TAG_USAGE

```
(TAG,VALUE,NEQTAG,COMPNAME,CURR_VALUE,TYPEINFO)
```

XCHECK_UNION_TAG

```
(TAG,VALUE,ASSTAG,NEQTAG,COMPNAME,CURR_VALUE,
 TYPEINFO)
```

XCHECK_UNION_TAG_FREE

```
(TAG,VALUE,ASSTAG,NEQTAG,FREEFUNC,COMPNAME,
 CURR_VALUE,TYPEINFO)
```

The UNION macros are used to test tag in a union variable. The XCHECK_UNION_TAG and XCHECK_UNION_TAG_FREE check the tag when some component of the union is assigned a value. The

FREE version of the macro is used if the union contains some component that has a Free function. The XCHECK_UNION_USAGE is used to test if an accessed component is active or not.

Parameters:

- TAG
  The tag component
- VALUE
  The expected tag value
- ASSTAG
  The assignment function for the tag type
- NEQTAG
  The equal test function for the tag type
- FREEFUNC
  The Free function for the UNION type
- COMPNAME
  The name of the selected component as a char string
- CURR_VALUE
  The current value of the tag type
- TYPEINFO
  The type info node for the tag type.

XCHECK_REF

XCHECK_OWN

XCHECK_OREF

```
(VALUE,REF_TYPEINFO,REF_SORT)
```

These macros are used to implement a test that Null pointers (using the Ref, Own, or ORef generator) are not de-referenced. These macros are inserted before each statement containing a Ref/Own/ORef pointer de-referencing. In case of an ORef pointer it is also checked that the ORef is valid, i.e. that it refers to an object owned by the current process.

Parameters:

- VALUE
  This is the value of the pointer.

- REF_TYPEINFO
  The typeinfo node for the Ref sort.
- REF_SORT
  The C type that corresponds to the Ref instantiation newtype.

XCHECK_OREF2

```
(VALUE)
```

Checks that a ORef pointer is a valid pointer, i.e. NULL, or that it refers to an object owned by the current process.

Debug and Simulation Macros

XAFTER_VALUE_RET_PRDCALL

```
(SYMB_NO)
```

A macro generated between the implementation of a value returning procedure call (implicit call symbol) and the symbol containing the value returning procedure call. `SYMB_NO` is the symbol number of the symbol containing the value returning procedure call.

XAT_FIRST_SYMBOL

```
(SYMB_NO)
```

A macro generated between an input or start symbol and the first symbol in the transition. `SYMB_NO` is the symbol number of the first symbol in the transition.

XAT_LAST_SYMBOL

A macro generated immediately before a nextstate or stop operation.

XBETWEEN_STMTS

XBETWEEN_STMTS_PRD

```
(SYMB_NO, C_LINE_NO)
```

A macro generated between statements in a task. The suffix _PRD indicates that these statements are part of a procedure.

Parameters:

- `SYMB_NO`
  the symbol number of the next statement.

- C_LINE_NO
  line number in C of this statement.

XBETWEEN_SYMBOLS

XBETWEEN_SYMBOLS_PRD

(SYMB_NO, C_LINE_NO)

A macro generated between symbols in a transition. The suffix _PRD indicates that these symbols are part of a procedure.

Parameters:

- SYMB_NO
  the symbol number of the next symbol.

- C_LINE_NO
  line number in C of this statement.

XDEBUG_LABEL

(LABEL_NAME)

This macro gives the possibility to insert label at the beginning of transitions. Such labels can be useful during debugging. The LABEL_NAME parameter is a concatenation of state name and the signal name. The * in state *; and input *; will cause the name ASTERISK to appear.

XOS_TRACE_INPUT

(SIG_NAME_STRING)

This macro is generated at input statements and can, for example, be used to generated trace information about inputs. The SIG_NAME_STRING parameter is the name of the signal in the input.

YPRSNAME_VAR

(PRS_NAME_STRING)

This macro is generated among the declarations of variables in the PAD function for a process. It can, for example, be used to declare a `char *` variable containing the name of the process. Such a variable can be useful during debugging. The PRS_NAME_STRING parameter is the name of the process as a character string.

YPRDNAME_VAR

(PRD_NAME_STRING)

This macro is generated among the declarations of variables in the PRD function for a procedure. It can, for example, be used to declare a `char*` variable containing the name of the procedure. Such a variable can be useful during debugging. The `PRD_NAME_STRING` parameter is the name of the procedure as a character string.

Utility Macros to Be Inserted

The following sequence of macros should be inserted. Most of them concern removal of struct components (in IdNodes) that are not used due to the combination of other switches used.

```
#define NIL 0
#define XXFREE xFree
#define XSYSD xSysD.

#if defined(XPRSPRIO) || defined(XSIGPRSPRIO) ||
    defined(XPRSSIGPRIO)
#define xPrsPrioPar(p)  , p
#else
#define xPrsPrioPar(p)
#endif

#if defined(XSIGPRIO) || defined(XSIGPRSPRIO) ||
    defined(XPRSSIGPRIO)
#define xSigPrioPar(p)  , p
#define xSigPrioParS(p)  p;
#else
#define xSigPrioPar(p)
#define xSigPrioParS(p)
#endif

#ifdef XTESTF
#define xTestF(p)  , p
#else
#define xTestF(p)
#endif

#ifdef XREADANDWRITEF
#define xRaWF(p)  , p
#else
#define xRaWF(p)
#endif

#ifdef XFREEFUNCS
#define xFreF(p)  , p
#else
#define xFreF(p)
#endif

#ifdef XFREESIGNALFUNCS
#define xFreS(p)  , p
```

```
#else
#define xFreS(p)
#endif

#define xAssF(p)
#define xEqF(p)



#ifdef XIDNAMES
#define xIdNames(p)  , p
#else
#define xIdNames(p)
#endif

#ifndef XOPTCHAN
#define xOptChan(p)  , p
#else
#define xOptChan(p)
#endif

#ifdef XBREAKBEFORE
#define xBreakB(p)  , p
#else
#define xBreakB(p)
#endif

#ifdef XGRTRACE
#define xGRTrace(p)  , p
#else
#define xGRTrace(p)
#endif

#ifdef XMSCE
#define xMSCETrace(p)  , p
#else
#define xMSCETrace(p)
#endif

#ifdef XTRACE
#define xTrace(p)  , p
#else
#define xTrace(p)
#endif

#ifdef XCOVERAGE
#define xCoverage(p)  , p
#else
#define xCoverage(p)
#endif

#ifdef XNRINST
#define xNrInst(p)  , p
#else
#define xNrInst(p)
```

```
#endif

#ifdef XSYMBTLINK
#define xSymbTLink(p1, p2) , p1, p2
#else
#define xSymbTLink(p1, p2)
#endif

#ifdef XEVIEW
#define xeView(p)  p,
#define xeViewS(p)  p;
#else
#define xeView(p)
#define xeViewS(p)
#endif

#ifdef XCTRACE
#define xCTrace(p)  p,
#define xCTraceS(p)  p;
#else
#define xCTrace(p)
#define xCTraceS(p)
#endif

#ifndef XNOUSEOFSERVICE
#define xService(p)  , p
#else
#define xService(p)
#endif

#if !defined(XPMCOMM) && !defined(XENV)
#define xGlobalNodeNumber() 1
#endif

#define xSizeOfPathStack 50

#ifndef xOffsetOf
#define xOffsetOf(type, field) \
        ((xptrint) &((type *) 0)->field)
#endif
#define xToLower(C) \
        ((C >= 'A' && C <= 'Z') ? \
        (char)((int)C - (int)'A' + (int)'a') : C)

#ifndef xDefaultPrioProcess
#define xDefaultPrioProcess      100
#endif

#ifndef xDefaultPrioSignal
#define xDefaultPrioSignal       100
#endif

#ifndef xDefaultPrioTimerSignal
#define xDefaultPrioTimerSignal  100
#endif
```

```
#ifndef xDefaultPrioContSignal
#define xDefaultPrioContSignal   100
#endif

#ifndef xDefaultPrioCreate
#define xDefaultPrioCreate       100
#endif

#define xbool int

#ifndef MAX_READ_LENGTH
#define MAX_READ_LENGTH 5000
                        /* max length of input line */
#endif
```

The xDefaultPrio macros above should, of course, be defined to the suitable default values.

Other macros that should be defined are.

SDL_NULL

a null value for the type PId.

xNotDefPId

which is used as RECEIVER parameter in the SDL_2OUTPUT macros. Please see also the section were signals are treated.

Chapter

62 *The ADT Library*

This chapter provides information about the library of Abstract Data Types (ADT) that comes with the SDL Suite. The data types provide services that are often needed in SDL systems.

The ADT library is mainly intended for usage together with the Cadvanced/Cbasic SDL to C Compiler and the ordinary simulation, validation, and applications kernels. Some of the ADTs are, however, also possible to use together with OS integrations (Cadvanced) and with Cmicro. If this is the case it is indicated in the description of the ADT.

General

The ADT library currently contains the following:

- A package, `ctypes`, that contains a number of sorts and generators to simplify an integration with C data types
- A data type that provides handling of text files and I/O operations as SDL operator calls
- A data type to generate random numbers from a number of distributions
- A data type that implements linked lists
- Data types for byte, unsigned, long int and so on (provided only for backward compatibility; use package `ctypes` instead)
- A data type that makes it possible to define PId literals for static process instances as synonyms
- A data type that provides a number of general purpose operators that may be used to reduce the complexity of an SDL system

These data types are delivered in source code. Feel free to change and adapt these data types for your own needs.

Important!

There is no commitment from IBM Rational to support the ADTs described in this chapter. IBM Rational has used the ADTs in internal projects with successful results.

The files that are contained in the ADT library are located in the subdirectory `<installation directory>/include/ADT`. (**In Windows**, replace `/` in the path above with `\`)

Note: Conformance with earlier releases

The ADTs in SDL Suite are backward compatible with the ADTs in earlier releases, in the sense that you only need to include the new versions of the ADTs to obtain the same behavior.

However, it is important to remember that the ADTs and the code generators you use, must be from the same version of SDL Suite.

Integration with C Data Types

The package `ctypes` presented below contains a number of types and generators that is intended to directly support C data types in SDL. The package `ctypes` can also be used in OS integrations and with Cmicro. However, usage of C pointers (generator Ref) might cause problems, due to potential memory leaks and potential memory access protection between OS tasks.

The file `ctypes.sdl` is a SDL/PR version of this package suitable to use in an include statement in an SDL/PR system, while `ctypes.sun` is a SDL/GR version of the package.

In an SDL/GR system it is only necessary to insert a use clause, i.e.

```
use ctypes;
```

at a proper place. The Organizer will then by itself include the `ctypes` package, for example when the system is to be analyzed. To use the `ctypes` package in an SDL/PR system the following structure should be used.

```
/*#include 'ctypes.sdl'*/

use ctypes;
system example;
  ...
endsystem;
```

The `ctypes` package consists of the following newtypes, syntypes, and generators:

SDL	C
`syntype ShortInt`	`short int, short`
`syntype LongInt`	`long int, long`
`syntype UnsignedShortInt`	`unsigned short int, unsigned short`
`syntype UnsignedInt`	`unsigned int, unsigned`
`syntype UnsignedLongInt`	`unsigned long int, unsigned long`
`syntype Float`	`float`
`newtype Charstar`	`char *`

SDL	C
newtype Voidstar	void *
newtype Voidstarstar	void **
generator Carray	array type
generator Ref	pointer type

All the newtypes and syntypes introduce type names for “standard types” in C.

Some of the types and generators are briefly described below.

Charstar

In Charstar there is a literal and some operators included:

```
LITERALS
  Null;
OPERATORS
  cstar2cstring : Charstar   -> Charstring;
  cstring2cstar : Charstring -> Charstar;
  cstar2vstar   : Charstar   -> Voidstar;
  vstar2cstar   : Voidstar   -> Charstar;
  cstar2vstarstar : Charstar -> Voidstarstar;
```

Note that the operators `cstar2cstring` and `cstring2cstar` are not available when using Cmicro.

The operators are all conversion routines to convert a value from one type to another. Note that Charstar and Charstring are **not** the same types even if they both corresponds to char * in C. Note also that freeing allocated memory for Charstar is the responsibility of the user, as there is not enough information to handle this automatically (as for Charstring). For more information about how to free memory, see the Ref generator below.

Voidstarstar

The Voidstarstar type has all the properties of the Ref generator (see below). This means that `*>`, `&`, `+`, and `-` can be used and that the following literal and operators are defined:

```
LITERALS
  Null,
  Alloc;
OPERATORS
  vstarstar2vstar : Voidstarstar -> Voidstar;
  vstar2vstarstar : Voidstar -> Voidstarstar;
```

Carray

The generator Carray has the following parameters:

```
GENERATOR Carray (CONSTANT Length, TYPE Itemsort)
```

where Length is an integer giving the number of elements of the array (index from 0 to Length-1), and Itemsort gives the type of each element in the array. A Carray in SDL is translated to an array in C. Indexing a Carray variable in SDL follows the same rules as for ordinary SDL Arrays.

Ref

The generator Ref has the following definition:

```
GENERATOR Ref (TYPE Itemsort)
  LITERALS
    Null,
    Alloc;
  OPERATORS
    "*>"       : Ref, Itemsort -> Ref;
    "*>"       : Ref -> Itemsort;
    "&"        : Itemsort -> Ref;
    make!      : Itemsort -> Ref;
    free       : in/out Ref;
    "+"        : Ref, Integer -> Ref;
    "-"        : Ref, Integer -> Ref;
    Ref2VStar  : Ref -> Voidstar;
    VStar2Ref  : Voidstar -> Ref;
    Ref2VStarStar : Ref /*#REF*/ -> Voidstarstar;
  DEFAULT Null;
ENDGENERATOR Ref;

procedure Free; fpar p Voidstarstar;
external;
```

Instantiating the Ref generator creates a pointer type on the type given as generator parameter. The literals and operators have the following behavior:

- **Null**: The NULL value (= 0) for the pointer type. This is also the default value for a pointer variable.

- **Alloc**: An operator without parameters that returns a new allocated data area with the size of the Itemsort.
- ***>**: This is the extract! and modify! operator and can be used to reference the value referenced by a pointer. If p is a pointer type, `p*>` is the value the pointer refers to. Comparing with C, `p*>` is the same as `*p`. If p is a pointer to a struct, then `p*>!a` is the same as `(*p).a`.
- **&**: The & operator corresponds to the C operator with the same name. It can be used to take the address of a variable. Comparing with C, &p and &(p) in SDL is the same as &p in C.
- **make!**: The make! operator, which as usual in SDL has the syntax (. .), is a short hand for creating memory and initializing it to a given value. The statement:

  ```
  a := (. 2 .);
  ```

  has the same meaning as

  ```
  a := Alloc, a*> := 2;
  ```

- **free**: The Free operator is used to deallocate memory referenced by a Ref pointer. If the component type contains automatically handled pointers (Charstring, Octet_string, Bit_string, Bags, Own pointers, and so on) the memory for these components is also deallocated.
- **+, -**: These operator have the meaning of pointer arithmetics in exactly the same way as in C. For example, p+1 (if p is of a pointer type) will add the Itemsort size to the value p. The + and - operators are mostly used to step through an array in C.
- **ref2vstar**, **vstar2ref**, **ref2vstarstar**: These operators are conversion operators, that can be used to cast between pointers and void * and void **.
- procedure **Free**: NOTE: Old feature provided for backward compatibility. Use operator free above instead.

  This external procedure is closely connected to the Ref generator. It should be used to deallocate memory allocated by the Alloc operator. Free should be passed the pointer variable that references the data area to be released. The variable should be casted to Voidstarstar. After calling Free the pointer variable will have the value Null.

  Example: `Free(Ref2VStarStar(variable_name))`

  Apart from the difficult syntax for calling the Free procedure it has another problem, it does not free components inside the referenced data area as the free operator above does.

The SDL Analyzer can allow implicit type conversion of pointer data types created by the Ref generator; see "Implicit Type Conversions" on page 134 in chapter 3, *Using SDL Extensions, in the SDL Suite 6.2 Methodology Guidelines*.

Abstract Data Type for File Manipulations and I/O

The ADT TextFile

In this section an SDL abstract data type, `TextFile`, is discussed where file manipulations and I/O operations are implemented as operations on the abstract data type. This ADT can be used also in OS integrations and in Cmicro if the target system has support for files in C.

This data type, which you may include in any SDL system, makes it possible to access, at the SDL level, a subset of the file and I/O operations provided by C.

The implementation of the operators are harmonized with the I/O in the monitor, including the Simulator Graphical User interface. All terminal I/O, for example, will be logged on the interaction log file if the monitor command Log-On is given.

The data type defines a "file" type and contains three groups of operations:

1. Operations to open and close files
2. Operations to write information onto a file
3. Operations to read information from a file.

The operations may handle I/O operations both on files and on the terminal (file `stdin` and `stdout` in C).

Note:

This data type is not intended to be used in the SDL Explorer!

Purpose

The `TextFile` data type supplies basic file and I/O operations as abstract data type operations in SDL, whereby I/O may be performed within the SDL language. The operations may handle I/O both on the

terminal and on files and are harmonized with the I/O from the monitor, from the trace functions, and from the functions handling dynamic errors.

To make the data type available you include the file containing the definition with an analyzer include in an appropriate text symbol:

Example 534: Including an ADT File ––––––––––––––––––––––––––

```
/*#include 'file.pr' */
```

––

Remember that all file systems are operating system specific. Any rules in your file system apply.

Summary of Operators

The following literals are available in the data type FileName:

```
SYNTYPE FileName = Charstring
ENDSYNTYPE;

SYNONYM NULL    FileName = 'NULL';
SYNONYM stdin   FileName = 'stdin';
SYNONYM stdout  FileName = 'stdout';
SYNONYM stderr  FileName = 'stderr';
```

The following literals and operators are available in the data type `TextFile`:

```
NEWTYPE TextFile
  LITERALS
    NULL, stdin, stdout, stderr;

  OPERATORS
    GetAndOpenR   : FileName -> TextFile;
    GetAndOpenW   : FileName -> TextFile;
    OpenR         : FileName -> TextFile;
    OpenW         : FileName -> TextFile;
    OpenA         : FileName -> TextFile;
    Close         : TextFile -> TextFile;
    Flush         : TextFile -> TextFile;
    IsOpened      : TextFile -> Boolean;
    AtEOF         : TextFile -> Boolean;
    AtLastChar    : TextFile -> Boolean;

    PutReal       : TextFile, Real -> TextFile;
    PutTime       : TextFile, Time -> TextFile;
    PutDuration   : TextFile, Duration -> TextFile;
    PutPId        : TextFile, PId -> TextFile;
    PutInteger    : TextFile, Integer -> TextFile;
    PutBoolean    : TextFile, Boolean -> TextFile;
```

```
    PutCharacter  : TextFile, Character -> TextFile;
    PutCharstring : TextFile, Charstring -> TextFile;
    PutString     : TextFile, Charstring -> TextFile;
    PutLine       : TextFile, Charstring -> TextFile;
    PutNewLine    : TextFile -> TextFile;
    "//"          : TextFile, Real -> TextFile;
    "//"          : TextFile, Time -> TextFile;
    "//"          : TextFile, Duration -> TextFile;
    "//"          : TextFile, Integer -> TextFile;
    "//"          : TextFile, Charstring -> TextFile;
    "//"          : TextFile, Boolean -> TextFile;
    "//"          : TextFile, PId -> TextFile;
    "+"           : TextFile, Character -> TextFile;

    GetReal       : TextFile, Charstring -> Real;
    GetTime       : TextFile, Charstring -> Time;
    GetDuration   : TextFile, Charstring -> Duration;
    GetPId        : TextFile, Charstring -> PId;
    GetInteger    : TextFile, Charstring -> Integer;
    GetBoolean    : TextFile, Charstring -> Boolean;
    GetCharacter  : TextFile, Charstring -> Character;
    GetCharstring : TextFile, Charstring -> Charstring;
    GetString     : TextFile, Charstring -> Charstring;
    GetLine       : TextFile, Charstring -> Charstring;
    GetSeed       : TextFile, Charstring -> Integer;

    GetReal       : TextFile -> Real;
    GetTime       : TextFile -> Time;
    GetDuration   : TextFile -> Duration;
    GetPId        : TextFile -> PId;
    GetInteger    : TextFile -> Integer;
    GetBoolean    : TextFile -> Boolean;
    GetCharacter  : TextFile -> Character;
    GetCharstring : TextFile -> Charstring;
    GetString     : TextFile -> Charstring;
    GetLine       : TextFile -> Charstring;
    GetSeed       : TextFile -> Integer;
ENDNEWTYPE TextFile;
```

The operators may be divided into three groups with different purpose:

1. Operators that, together with the literals, are used for handling files.
2. Operators suited for writing information to files.
3. Operators intended for reading information from files.

The next three subsections provide the necessary information for using these operators. The data type itself will be discussed together with the operators for handling files.

File Handling Operators

First in this subsection each operator and literal will be discussed in detail and then some typical applications of the operators will be presented.

> **Caution!**
>
> The operators GetAndOpenR and GetAndOpenW **do not work** with the Application library. The operators GetPId and PutPId (and the // operator to write PIds) can be used with the Application library, but they will use a different output format.

Operator Behavior

The type `TextFile` is implemented using the ordinary C file type `FILE`. A `TextFile` is a pointer to a `FILE`.

```
typedef FILE * TextFile;
```

The literal `NULL` represents a null value for files. This literal is translated to `TextFileNull()` in the generated C code by an appropriate `#NAME` directive and is then implemented using the macro:

```
#define TextFileNull() (TextFile)0
```

All variables of the type `TextFile` will have this value as default value.

The literals `stdin` and `stdout` represent the standard files `stdin` and `stdout` in C, which are the files used in C for I/O to the terminal. The file `stdin` is used for reading information from the keyboard, while `stdout` is used for writing information on the screen.

The standard operators assignment and test for equality is implemented in such a way that `A:=B` means that now A refers to the same file as B, while `A=B` tests if A and B refer to the same file.

FileName

The data type `FileName` is used to represent file names in the operators `GetAndOpenR`, `GetAndOpenW`, `OpenR`, `OpenW`, and `OpenA`. It has all Charstring literals and the special synonyms `NULL`, `stdin` (input from the keyboard), `stdout` (output to the screen), and `stderr` (output to the screen from which the SDL Suite was started). As `FileName` is a syn-

type of Charstring, the usual Charstring operators are defined for this type.

Caution!

The synonyms `stdin`, `stdout`, `stderr` in some circumstances hide the literals with the same names according to SDL scope rules. If that is the case, please insert a qualifier `<<type textfile>>` before the literal name.

GetAndOpenR – GetAndOpenW

The operators `GetAndOpenR` and `GetAndOpenW` are used to open a file with a name prompted for on the terminal. `GetAndOpenR` opens the file for read, while `GetAndOpenW` opens the file for write. The operators take the prompt as parameter (type charstring), print the prompt on the screen (on `stdout`), and read a file name from the keyboard (from `stdin`). An attempt is then made to open a file with that name. If the open operation was successful, a reference to the file is returned by the `GetAndOpenR` or `GetAndOpenW` operator, otherwise `NULL` is returned. After a successful open operation you may use the file for reading or writing.

If you type `<Return>`, `-` or the file name `stdin` at the prompt in `GetAndOpenR` a reference to `stdin` is returned by the operator. `GetAndOpenW` will, in the same way, return a reference to `stdout` if the prompt is answered by `<Return>`, `-` or the file name `stdout`.

Note:

To work properly in the *Simulator Graphical User Interface*, the prompt string should be terminated with: ": ", i.e. colon space.

OpenR – OpenW – OpenA

The operators `OpenR`, `OpenW`, and `OpenA` are used to open a file with a file name passed as parameter. `OpenR` opens the file for read, while `OpenW` opens the file for write and `OpenA` opens the file for append. An attempt is made to open a file with the name given as a parameter. If the open operation was successful, a reference to the file is returned by the `OpenR`, `OpenW`, or `OpenA` operator, otherwise `NULL` is returned. After a successful open operation you may read, write or append on the file.

Close

The operator `Close` is used to close the file passed as parameter. `Close` always returns the value `NULL`. This operator should be used on each file opened for write after all information is written to the file to ensure that any possibly buffered data is flushed.

> **Note:**
>
> Always close a file variable before assigning it to a new file, otherwise data may be lost.

Flush

Output to files is usually buffered, and is therefore not immediately written on the physical output device. The operator `Flush` forces the output buffer of the file that is passed as parameter to be written on the physical output device. It is equivalent to C function `fflush`.

IsOpened

The operator `IsOpened` may be used to determine if a `TextFile` is open or not. It may, for example, be used to test the result of the `Open` operation discussed above. The test `IsOpened(F)` is equivalent to `F /= NULL`.

AtEOF

The operator `AtEof` may be used to determine if a `TextFile` has reached the end of file or not. This operator could be used in order to determine when to stop reading input from a file. The test `AtEof(F)` is equivalent to `feof(F)`.

> **Note:**
>
> `AtEof` first becomes true when attempts are made to read behind the end-of-file. Operator `AtLastChar` becomes true when the last character of the file has been read, and is usually more useful than `AtEof`.

AtLastChar

The operator `AtLastChar` may be used to determine if a `TextFile` has reached the end of file or not. This operator is useful in order to determine when to stop reading input from a file. The test `AtLastChar(F)` returns `true` if there are no more characters to be read from the file.

Examples of Use

Three typical situations when you want to write information are easily identified:

1. The information is to be printed on the screen.
2. The information is to be printed on a file with a given name.
3. You want to determine at run-time where the information is to be printed.

Example 535: ADT for File I/O, Print to Screen ––––––––––––––––––

If the information is to be **printed on the screen**, you may use the following structure:

```
DCL F TextFile;
TASK F := stdout // 'example';
```

Declare a variable of type `TextFile` and assign it the value `stdout`. You may then use it in the write operators discussed under "Write Operators" on page 3229.

––

Example 536: ADT for File I/O, Print to File ––––––––––––––––––––

If the information is to be **printed on a file** with a given name, you may use the following structure:

```
DCL F TextFile;
TASK F := OpenW('filename');
TASK F := F // 'example';
```

The difference from the above is that the operator `OpenW` is used to open a file with the specified name. This outline may be complemented with a test if the `OpenW` operation was successful or not.

––

Example 537: ADT for File I/O, Accessing Text File ––––––––––––––––

If you **want to be able to determine at run-time where the information should be printed**, you should define a `TextFile` as in the examples above, and then use the following structure.

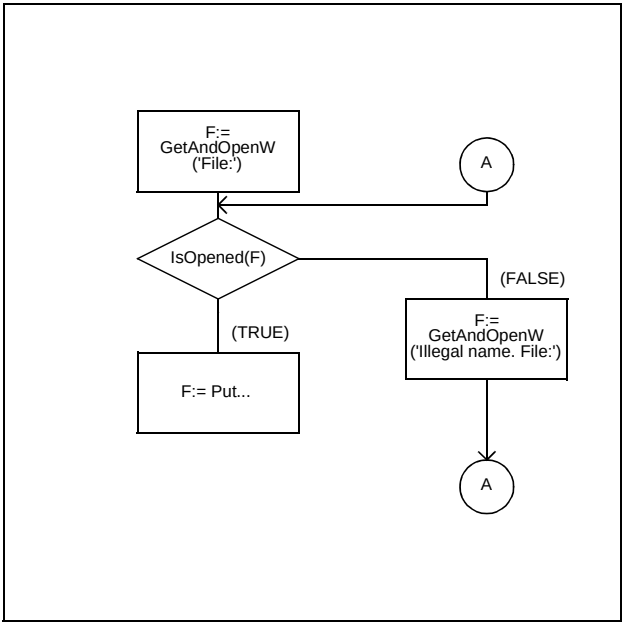


Figure 563: Accessing a TextFile

If you answer the question by hitting `<Return>` or by typing `stdout`, the information will be printed on screen (`stdout`). If you type the name of a file, the information will be printed on that file.

––

If you want to open the file for read instead of write, you may use almost identical structures.

Write Operators

Operator Behavior

The write operators `PutReal`, `PutTime`, `PutDuration`, `PutInteger`, `PutBoolean`, `PutCharacter`, and `PutCharstring` all take a `TextFile` and a value of the appropriate type as parameters. The operators print the value passed as parameter on the file referenced by the `TextFile` parameter and then return the `TextFile`. The *Put** operators will print the values in the same format as the monitor uses for the command Examine-Variable, and will append a space after each printed value.

The operator `PutString` takes a `TextFile` and a `Charstring` parameter and prints the string on the `TextFile`. `PutString` prints the string as a C string, not using the format for SDL Charstring. This means that no ' is printed. `PutString` returns the `TextFile` given as parameter as result.

The infix write operator `"//"` takes as parameters a `TextFile` and a value of type `Boolean, Charstring, Integer, PId, Real, Time,` or `Duration`. `TextF // Val` prints the value `'Val'` to the `TextFile` referenced by `'TextF'`, and returns value `'TextF'`. `Character` strings are printed without enclosing `''`. All // operators except the one for `Charstring` append a space to the file, after the value is written.

The infix write operator `"+"` takes as parameters a `TextFile` and a `Character`. `"+"` behaves just as `"//"`, but it has its special name in order to avoid type conflicts with `Charstring`.

The operator `PutNewLine` takes a `TextFile` as parameter, prints a carriage return (actually a `"\n"`) on this file, and returns the `TextFile` as operator result.

The different `Put` operators are equivalent to the `//` operators, and they are mainly present for backward compatibility reasons.

There is a function named `xPutValue` in the implementation of the data type `TextFile`. This function may print a value of any type that may be handled by the monitor system, but may only be accessed from in-line C code and not from SDL. A detailed description of the `xPutValue` function may be found under "Accessing the Operators from C" on page 3231.

Example 538: ADT for File I/O, Print to File ––––––––––––––––––––

To print a line according to the following example, where 137 is the value of the variable `NoOfJobs`:

```
Number of jobs: 137 Current time: 137.0000
```

You could use the following statements, assuming that the `TextFile F` is already opened:

```
TASK
  F := F // 'Number of jobs: ' // NoOfJobs;
TASK
  F := F // 'current time: ' // Now;
TASK
  F := PutNewLine(F);
```

––

Read Operators

Operator Behavior

The read operators `GetReal`, `GetTime`, `GetDuration`, `GetInteger`, `GetBoolean`, `GetCharacter`, `GetCharstring`, and `GetSeed` are used to read values of the various sorts.

The operator `GetSeed` is used to read appropriate values to initialize random number generators (odd integers in the range 1 to 32767).

There are two versions of each `Get` operator: one that only takes as parameters a `TextFile`, and the other that takes as parameters a `TextFile` and a `Charstring` which is used as prompt. All `Get` operators behave differently depending on if the value should be read from the terminal (`stdin`) or from a file.

If the value should be **read from the terminal**, the `Get` operators with the prompt parameter may be used. This prompt is printed on the screen, and then an attempt to read a value of the current type is made. If a `Get` operator with only the `TextFile` parameter is used, a default prompt is used, that depends on the type that is to be input. If the operation is successful, the read value is returned as a result, otherwise the message `"Illegal value"` is printed and the user is given a new chance to type a value.

Note:

To work properly in the *Simulator Graphical User Interface*, the prompt string should be terminated with: ": ", i.e. colon space.

If the value should be **read from a file**, it is recommended to use the `Get` operators without the prompt parameter, as it is not used. It is assumed that a value of the correct type will be found.

There are several `Get` operators for reading character strings:

`GetString` reads a sequence of characters until the first white space character, and is equivalent to `fscanf (f, "%s")`.

`GetLine` reads a sequence of characters until the end of line is reached. It is equivalent to `fgets`, but the end-of-line character will not be part of the string.

`GetCharstring` reads a sequence of characters on a single line that is enclosed by single quotes `(')`. This operator is mainly present for backward compatibility reasons.

There is a function named `xGetValue` in the implementation of the data type `TextFile`, which may read a value of any type that may be handled by the monitor system. This function can only be accessed from in-line C code and not from SDL. A detailed description of the `xGetValue` function may be found under "Accessing the Operators from C" on page 3231.

Example 539: ADT for File I/O, Read from File

```
TASK
  Mean := GetReal (F),
  A(1) := GetReal (F),
  A(2) := GetReal (F),
  A(3) := GetReal (F);
```

Accessing the Operators from C

In some circumstances it may be easier to use C code (in `#CODE` directives) rather than SDL to implement an algorithm. SDL implementations for linear algorithms sometimes become unnecessarily large and complex, as SDL for example lacks a loop concept. Consider the SDL

graph in Figure 563 on page 3228. This graph could be replaced by a TASK with the following contents:

```
'Open file F' /*#CODE
#(F) = GetAndOpenW("LFile : ");
while ( ! IsOpened(#(F)) )
  #(F) = GetAndOpenW("LIllegal name. File : "); */
```

which is more compact and gives a better overview at the SDL level.

#(F) is an SDL directive telling the SDL to C compiler to translate the SDL variable F to the name it will receive in the generated C code.

To simplify the use of in-line C code, #NAME directives are introduced on all identifiers defined in this data type. The same names are used in C as in SDL.

Note:

Upper and lower case letters are significant in C (but not in SDL).

Note also the additional L in the Charstring literals, for example "LIllegal name. File : ". This first character is used in the implementation of the SDL sort Charstring and should **always** be L in a charstring literal.

From in-line C, two functions xGetValue and xPutValue are also available to read and write values of any type. These functions have the following prototypes:

```
extern void xGetValue(
  TextFile       F,
  SDL_Charstring Prompt,
  xSortIdNode    SortId,
  void         * Result,
  char         * FunctionName );

extern void xPutValue(
  TextFile    F,
  xSortIdNode SortId,
  void      * Value,
  char      * FunctionName );
```

Parameter	Interpretation
TextFile F	The file to read from or to print to.
In xGetValue: SDL_Charstring Prompt	Is used as prompt in exactly the same way as for the ordinary Get operators.

Parameter	Interpretation
`xSortIdNode SortId`	A reference to the `xIdNode` that represents the SDL sort to be read or printed For the **predefined SDL sorts** you may use variables named `xSrtN_SDL_Real`, `xSrtN_SDL_Integer`, and so on, as parameter. For **user-defined sorts**, you may use similar variables named `ySrtN_#(SortName)`, where `SortName` should be replaced by the sort name.
`void * Result` `void * Value`	The address to the variable where the result should be stored, or the address to the variable that should be printed.
`char * FunctionName`	A string specifying the name of an appropriate function. This name will be given if an error is detected during reading or printing.

Note:

xGetValue and xPutValue **will not work** together with the Application library.

To handle, for example, I/O of an SDL struct, the ideas presented below may be used.

Example 540: ADT for File I/O of an SDL Struct

```
NEWTYPE SName STRUCT
  a, b Integer;
ENDNEWTYPE;

DCL
  FIn, FOut TextFile,
  SVar SName;

TASK 'Put SVar on FOut' /*#CODE
  xPutValue( #(FOut), ySrtN_#(SName),
  &(#(SVar)), "PutSName" ); */;

TASK 'Get SVar from FIn' /*#CODE
  xGetValue( #(FIn), "Value : ",
  ySrtN_#(SName),&(#(SVar)), "LGetSName"); */;
```

Abstract Data Type for Random Numbers

One important feature, especially in performance simulations, is the possibility to generate random numbers according to a number of distributions, like for example the negative exponential distribution and the Erlang distribution. It is also important that the random number sequences are reproducible, to be able to run a slightly modified version of a simulation with the same sequence of random numbers.

In this section an SDL abstract data type according to the previous discussion is presented. This data type may be included in any SDL system. This ADT can also be used in OS integrations and in Cmicro. It is, however, necessary to check that the typedef for the RandomControl, see below, refers to an unsigned 32-bit type.

Purpose

The SDL `RandomControl` data type allows you to generate pseudo-random numbers. A number of distributions are supported, including the negative exponential distribution and the Erlang distribution. In performance simulations, which is the main application area for this data type, the most important need for random numbers is in connection with `Time` and `Duration` values. It is, for example, interesting to draw inter- arrival times in job generators, and service lengths in servers. Distributions returning positive real numbers are thus most meaningful.

The basic mechanism behind pseudo-random number generation is as follows. A sequence of bit-patterns is defined using a formula of type:

$$Seq_{n+1} = f(Seq_n)$$

The function f should be such that the sequence of elements seen as numbers should be “random”, and the number of element in the sequence, before it starts to repeat itself, should be as large as possible.

To obtain a new random number is thus a two step process:

1. Compute and store a new bit-pattern from the old bit-pattern
2. Interpret the new bit-pattern as a number, which is returned as the new random number.

In this data type, 32 bit patterns, implemented in C using the type unsigned long, are used together with the formula:

$$Seq_{n+1} = ((2^{16} + 3) \cdot Seq_n) \bmod 2^{32}$$

The result returned by the operator `Random`, which is the basic random number generator, is this bit-pattern seen as a number between 0 and 1, and expressed as a float.

The data type `RandomControl` may be included in any SDL system using an analyzer include statement, where the file containing the definition of the data type is included. Example:

```
/*#include 'random.pr' */
```

As the C standard functions `log` and `exp` are used in the `random.pr` file, it is necessary to link the application together with the library for math functions, i.e. to have `-lm` in the link operation in the makefile. See "Makefile Options" on page 123 in chapter 2, *The Organizer*. To use the entry `-lm` in the link list seems to be a fairly standard way to find the library for math functions. If this does not work, or you want more details, please see the documentation for your C compiler.

Available Operators

The type `RandomControl`, introduced by this data type, is in C implemented as the address of an unsigned long.

```
typedef unsigned long * RandomControl;
```

Note that you have check that `unsigned long` is a 32 bit type, otherwise you have to change the typedef.

The reason for passing the address to the bit-pattern (that is to the unsigned long), is that this bit-pattern has to be updated by the random functions.

Below the operators provided in this data type are listed. There are, for many of the operators, several versions with different sets of parameters and/or result types to support different usage of the operator.

```
Random : RandomControl -> Real;
Random : RandomControl -> Duration;
Random : RandomControl -> Time;

Erlang : Real, Integer, RandomControl -> Real;
Erlang : Real, Integer, RandomControl -> Duration;
Erlang : Real, Integer, RandomControl -> Time;
Erlang : Duration, Integer, RandomControl -> Real;
Erlang : Duration, Integer, RandomControl -> Duration;
Erlang : Duration, Integer, RandomControl -> Time;
Erlang : Time, Integer, RandomControl -> Real;
Erlang : Time, Integer, RandomControl -> Duration;
Erlang : Time, Integer, RandomControl -> Time;
```

```
HyperExp2 :
  Real, Real, Real, RandomControl -> Real;
HyperExp2 :
  Real, Real, Real, RandomControl -> Duration;
HyperExp2 :
  Real, Real, Real, RandomControl -> Time;
HyperExp2 :
  Duration, Duration, Real, RandomControl -> Real;
HyperExp2 :
  Duration, Duration, Real, RandomControl -> Duration;
HyperExp2 :
  Duration, Duration, Real, RandomControl -> Time;
HyperExp2 :
  Time, Time, Real, RandomControl -> Real;
HyperExp2 :
  Time, Time, Real, RandomControl -> Duration;
HyperExp2 :
  Time, Time, Real, RandomControl -> Time;

NegExp : Real, RandomControl -> Real;
NegExp : Real, RandomControl -> Duration;
NegExp : Real, RandomControl -> Time;
NegExp : Duration, RandomControl -> Real;
NegExp : Duration, Rando mControl -> Duration;
NegExp : Duration, RandomControl -> Time;
NegExp : Time, RandomControl -> Real;
NegExp : Time, RandomControl -> Duration;
NegExp : Time, RandomControl -> Time;

Uniform : Real, Real, RandomControl -> Real;
Uniform : Real, Real, RandomControl -> Duration;
Uniform : Real, Real, RandomControl -> Time;
Uniform : Duration, Duration, RandomControl -> Real;
Uniform :
  Duration, Duration, RandomControl -> Duration;
Uniform : Duration, Duration, RandomControl -> Time;
Uniform : Time, Time, RandomControl -> Real;
Uniform : Time, Time, RandomControl -> Duration;
Uniform : Time, Time, RandomControl -> Time;

Draw    : Real, RandomControl -> Boolean;

Geometric : Real, RandomControl -> Integer;
Geometric : Real, RandomControl -> Duration;
Geometric : Real, RandomControl -> Time;
Geometric : Duration, RandomControl -> Integer;
Geometric : Duration, RandomControl -> Duration;
Geometric : Duration, RandomControl -> Time;
Geometric : Time, RandomControl -> Integer;
Geometric : Time, RandomControl -> Duration;
Geometric : Time, RandomControl -> Time;

Poisson : Real, RandomControl -> Integer;
Poisson : Real, RandomControl -> Duration;
```

```
Poisson : Real, RandomControl -> Time;
Poisson : Duration, RandomControl -> Integer;
Poisson : Duration, RandomControl -> Duration;
Poisson : Duration, RandomControl -> Time;
Poisson : Time, RandomControl -> Integer;
Poisson : Time, RandomControl -> Duration;
Poisson : Time, RandomControl -> Time;

RandInt : Integer, Integer, RandomControl -> Integer;
RandInt : Integer, Integer, RandomControl -> Duration;
RandInt : Integer, Integer, RandomControl -> Time;

DefineSeed : Integer -> RandomControl;
GetSeed    : Charstring -> Integer;
Seed       : RandomControl -> Integer;
```

Random (RandomControl)

The operator `Random` is the basic random generator and is called by all the other operators. `Random` uses the formula

$$Seq_{n+1} = ((2^{16} + 3) \cdot Seq_n) \bmod 2^{32}$$

to compute the next value stored in the parameter of type `RandomControl`. The result from `Random` is a real random number in the interval 0.0 < Value < 1.0.

Erlang (Mean, N, RandomControl)

The operator Erlang provides random numbers from the Erlang-N distribution with mean `Mean`. The first parameter Mean should be > 0.0, and the second parameter N should be > 0.

HyperExp2 (Mean1, Mean2, Alpha, RandomControl)

The `HyperExp2` operator provides random numbers from the hyperexponential distribution. With probability `Alpha` it return a random number from the negative exponential distribution with mean `Mean1`, and with the probability `1-Alpha` it returns a random number from the negative exponential distribution with mean `Mean2`. `Mean1` and `Mean2` should be > 0.0, and `Alpha` should be in the range 0.0 <= `Alpha` <= 1.0.

NegExp (Mean, RandomControl)

The operator `NegExp` provides random numbers from the negative exponential distribution with mean `Mean`. `Mean` should be > 0.0.

Uniform (Low, High, RandomControl)

The operator `Uniform` is given a range `Low` to `High` and returns a uniformly distributed random number in this range.
`Low` should be <= `High`.

Draw (Alpha, RandomControl)

The `Draw` operator returns true with the probability `Alpha` and false with the probability `1-Alpha`. `Alpha` should be in the range 0.0 <= `Alpha` <= 1.0.

Geometric (p, RandomControl)

The operator `Geometric` returns an integer random number according to the geometric distribution with the mean p/(1-p). The parameter `p` should be 0.0 <= `p` < 1.0.

Caution!

Since the range of feasible samples from the distribution is infinite and the result type is integer, integer overflow may occur.

Poisson (m, RandomControl)

The operator `Poisson` returns an integer random number according to the Poisson distribution with mean `m`. The parameter `m` should be >= 0.0.

Caution!

Since the range of feasible samples from the distribution is infinite and the result type is integer, integer overflow may occur.

RandInt (Low, High, RandomControl)

This operator `RandInt` returns one of the values `Low`, `Low+1`, . . . , `High-1`, `High`, with equal probability. `Low` should be <= `High`.

DefineSeed (Integer) -> RandomControl

Each `RandomControl` variable, which is used as a control variable for a random generator, has to be initialized correctly so the first bit-pattern used by the basic random function is a legal pattern. This `DefineSeed`

operator takes an integer parameter, which should be an odd value in the range 1 to 32767, and creates a legal bit-pattern. This first value is usually referred to as the seed for the random generator. Using the same seed value, the same random number sequence is generated, which means that the random number sequences are reproducible.

Seed (RandomControl) -> Integer

The `Seed` operator returns random numbers that are acceptable as parameters to the operator `DefineSeed`. If many `RandomControl` variables are to be initialized, the `Seed` operator may be useful.

GetSeed (Prompt) -> Integer

The `GetSeed` operator, which is implemented in the data type `TextFile` (see "The ADT TextFile" on page 3221), may be used to read an integer value that is acceptable as parameter to the `DefineSeed` operator.

Using the Data Type

To use the abstract data type for random number generation you must:

- Include the definition of the data type using an analyzer include. Usually it is appropriate to include the data type in a text symbol in the system diagram.
- Define a suitable number of `RandomControl` variables, one for each random number sequence that is to be used.
- Initialize the `RandomControl` variables, either in the variable declaration or in a `TASK` often placed in the start transition of the process. The operator `DefineSeed` should be used to initialize a `RandomControl` variable.
- Use the `RandomControl` variables in appropriate random number operators.

> **Note:**
>
> SDL variables can only be declared in processes and will be local to the process instances.

To have global RandomControl variables you may, however, define synonyms of type RandomControl and use them in random generator operators.

Example 541: Using RandomControl, DefineSeed

```
SYNONYM Seed1 RandomControl =
  DefineSeed(GetSeed(stdin, 'Seed1 : '));

TASK Delay := NegExp(Mean1, Seed1);
```

This is correct according to SDL as operators only have IN parameters and therefore expressions are allowed as actual parameters. In C it is also an IN parameter and cannot be changed. But as a RandomControl value is an address it is possible to change the contents in that address.

The SDL to C Compiler will, for synonyms that cannot be computed at generation time, allocate a variable and initialize it according to the synonym definition at start-up time. Note that this will be performed before any transitions have been executed.

A typical application of RandomControl synonyms are together with the Seed operator. The Seed operator is used to generate values suitable to initialize RandomControl variables with.

Example 542: Using RandomControl, Seed

```
SYNONYM BasicSeed RandomControl =
  DefineSeed(GetSeed(stdin, 'Seed : '));

DCL S1 RandomControl :=
  DefineSeed(Seed(BasicSeed));
DCL S2 RandomControl :=
  DefineSeed(Seed(BasicSeed));
```

The variety of operators with the same name makes it possible to directly use operators in many more situations. This is called overloading of operators. If, for example, there were only the NegExp version:

```
NegExp : Real, RandomControl -> Real;
```

then explicit conversion operators would have been necessary to draw, for example, a Duration value from the negative exponential distribution. The code to draw a Duration value would then be something like:

```
RealToDuration(NegExp(Mean, Seq))
```

We have instead introduced several operators with the same name and purpose, but with different combinations of parameter types and result type. So for the `NegExp` operator discussed above, there is also a version:

```
NegExp : Real, RandomControl -> Duration;
```

which is exactly what we wanted.

There is, however, a price to be paid for having overloaded operators. It must be possible for the SDL Analyzer to tell which operator that is used in a particular situation. It then uses all available information about the parameters and what the result is used for. Consider Example 543 below.

Example 543: Overloaded Operator ––––––––––––––––––––––––––

```
TIMER T;
DCL
  Mean, Rand Real,
  D Duration,
  Seq RandomControl :=
    DefineSeed(GetSeed('Seed : '));

TASK Rand := NegExp(Mean, Seq);
TASK D := NegExp(Mean, Seq);
TASK D := NegExp(TYPE Real 1.5, Seq);

DECISION NegExp(Mean, Seq) >
         TYPE Duration 10.0;
  (true) : ....
  ELSE :
ENDDECISION;
SET (Now + NegExp(Mean, Seq), T);
```

- The first two applications of `NegExp` are no problem, as the parameter type is given by the type of the `Mean` variable, and the result type is given by the variable that result is assigned to.
- In the third `NegExp` call, the value 1.5 has to be given a qualifier, that is, `TYPE Real`, as the literal 1.5 may be of type `Real`, `Duration`, or `Time`.
- In the fourth example it is the result type that cannot be determined if the literal 10.0 was not given with a qualifier.
- In the fifth example the only + operator that takes `Time` as left parameter and returns `Time` (`SET` should have a `Time` value as first parameter) is:

```
"+" : Time, Duration -> Time;
```

defined in the sort `Time`. So, both the type for the parameter and the result are possible to determine for the `NegExp` operator in this example.

Most of these problems can be avoided by using `SYNONYMS` or variables instead of literal values. This is in most cases a better solution than to introduce qualifiers.

Example 544: Using SYNONYMS

If, for example, the synonyms:

```
SYNONYM MeanValue Real = 1.5;
SYNONYM Limit Duration = 10.0;
```

were defined, the third and fourth `NegExp` call would cause no problem:

```
TASK D := NegExp(MeanValue, Seq);
DECISION NegExp(Mean, Seq) > Limit;
  (true) : ....
  ELSE :
ENDDECISION;
```

Trace Printouts

Trace printouts are available for the functions in this abstract data type. By assigning a trace value greater or equal to nine (9) using the monitor command Set-Trace, each call to an operator in this data type causes a printout of the name of the operator.

> **Note:**
>
> Each operator returning a random number will call the basic operator `Random` at least once.

Accessing the Operators from C

The operator for random number generation may be used directly in C by using the name given in the appropriate `#NAME` directive. Please look at the `random.pr` file for the `#NAME` directives.

Abstract Data Types for List Processing

The abstract data types defined in this “package” are intended for processing of linked lists. Linked lists are commonly appearing in applications and are one of the basic data structures in computer science. With these data types, you can concentrate on using the lists and do not have to worry about the implementation details, as all list manipulations are hidden in operators in the data types.

> **Note:**
>
> This data type is **not** implemented in a way that makes it possible to be used in the SDL Explorer. It can be used in OS integrations, but it is **not** recommended, due to the risk for memory leaks.

Purpose

Definitions

A *queue* is a list in which the members are ordered. The ordering is entirely performed by the user. The available operations make it possible to access members of the queue and insert members into or remove members from any position. Furthermore, the operators suppress the implementation aspects. That is, the fact that the queue is implemented as a doubly linked list with a queue head. The operators also prevent the unwary user from trying to access, for instance, the successor of the last member or the predecessor of the first member.

The entities which may be members of a queue are called *object instances*. An object instance is a passive entity containing user defined information. This information is described in the *object description*.

In SDL these definitions are implemented using sorts called `Queue`, `ObjectInstance`, and `ObjectDescr`, where `ObjectDescr` should be defined by the user. `ObjectDescr` should have the structure given in the example below (Example 545).

The data types for list processing may be included in any SDL system using Analyzer `#include` statements, where the files containing the definitions of the data types are included. The definitions should be placed in the order given in the example below:

Example 545: Including ADT for List Processing ––––––––––––––––

```
/*#include 'list1.pr'*/
NEWTYPE ObjectDescr /*#NAME 'ObjectDescr'*/
  STRUCT
    SysVar SysTypeObject;
    /* other user defined components */
ENDNEWTYPE;
/*#include 'list2.pr'*/
```

––

The file `list1.pr` contains the definition of the sort `Queue` (and the help sorts `ObjectType` and `SysTypeObject`), while the file `list2.pr` contains the definition of the type `ObjectInstance`.

Available Sorts

When the data types for list processing are included, two new sorts, `Queue` and `ObjectInstance`, are mainly defined, together with the type `ObjectDescr` defined by the user. The user can declare variables of type `Queue` and type `ObjectInstance`, but should never declare a variable of type `ObjectDescr`.

Variables of the sorts `Queue` and `ObjectInstance` are references (pointers) to the representation of the queue or the object instance. In both sorts there is a null value, the literal `NULL`, which indicates that a variable refers to no queue or no object instance. The default value for `Queue` and `ObjectInstance` variables is `NULL`.

A variable of sort `ObjectInstance` can refer to a data area containing the components defined in the struct `ObjectDescr`. The example below shows how to manipulate these components.

Example 546: ADT for List Processing, Struct ObjectDescr––––––––

```
/*#include 'list1.pr'*/
NEWTYPE ObjectDescr /*#NAME 'ObjectDescr'*/
  STRUCT
    SysVar SysTypeObject;
    Component1  Integer;
    Component2  Boolean;
ENDNEWTYPE;
/*#include 'list2.pr'*/

DCL O1 ObjectInstance;

TASK
  O1 := NewObject;  /* see next section */
```

```
TASK
  O1!ref!Component1 := 23,
  O1!ref!Component2 := false;

TASK
  IntVar := O1!ref!Component1,
  BoolVar := O1!ref!Component2;
```

A component is thus referenced by the syntax:

```
ObjectInstanceVariable ! ref ! ComponentName
```

Caution!

You should never directly manipulate the component `SysVar` in the struct `ObjectDescr`. It contains information about if and how the object instance is inserted into a queue and should only be used by the queue handling operators.

Assignments and test for equality may be performed for queues and for object instances. The assignments:

```
Q1 := Q2;  O1 := O2;
```

mean that `Q1` now refers to the same queue as `Q2` and that `O1` now refers to the same object instance as `O2`. Assignment is thus implemented as copying of the reference to the queue (and not as copying of the contents of the queue). The same is true for object instances.

The test for equality is in the same way implemented as a test if the left and right hand expression reference the same queue or the same object instance (and not if two queue or object instances have the same contents).

Due to the order in which the sorts are defined, a component of sort `Queue` can be a part of the `ObjectDescr` struct, while components of type `ObjectInstance` cannot be part of `ObjectDescr`.

If you want several different types of objects in a queue, with different contents, the `#UNION` directive (see *"Union" on page 2668 in chapter 56, The Cadvanced/Cbasic SDL to C Compiler*) may be used according to the following example:

Example 547: Unions and Queues ----------

```
NEWTYPE Ob1 STRUCT
  Comp1Ob1 integer;
  Comp2Ob1 boolean;
ENDNEWTYPE;

NEWTYPE Ob2 STRUCT
  Comp1Ob2 character;
  Comp2Ob2 charstring;
ENDNEWTYPE;

NEWTYPE Ob /*#UNION*/ STRUCT
  Tag integer;
  C1  Ob1;
  C2  Ob2;
ENDNEWTYPE;

NEWTYPE ObjectDescr /*#NAME 'ObjectDescr'*/
  STRUCT
    SysVar SysTypeObject;
    U      Ob;
/*#ADT (X)*/
ENDNEWTYPE;
```

The components may now be reached using:

```
O1 ! ref ! U ! Tag
O1 ! ref ! U ! C1 ! Comp1Ob1
O1 ! ref ! U ! C2 ! Comp2Ob1
```

Available Operators

Operators in the Sort Queue

In the sort `Queue`, the following literals and operators are available:

```
null
NewQueue

Cardinal     : Queue -> Integer;
DisposeQueue : Queue -> Queue;
Empty        : Queue -> Boolean;
FirstInQueue : Queue -> ObjectInstance;
Follow       :
     Queue, ObjectInstance, ObjectInstance -> Queue;
IntoAsFirst  : Queue, ObjectInstance -> Queue;
IntoAsLast   : Queue, ObjectInstance -> Queue;
LastInQueue  : Queue -> ObjectInstance;
Member       : Queue, ObjectInstance ->  Boolean;
```

```
Precede      :
     Queue, ObjectInstance, ObjectInstance -> Queue;
Predecessor  : ObjectInstance -> ObjectInstance;
Remove       : ObjectInstance -> ObjectInstance;
Successor    : ObjectInstance -> ObjectInstance;
```

Operators in the Sort ObjectInstance

In the sort `ObjectInstance`, the following literals and operators are available:

```
null
NewObject

DisposeObject: ObjectInstance -> ObjectInstance;
```

The operators defined in the sorts `Queue` and `ObjectInstance` have the behavior described below. All operators will check the consistency of the parameters. Each queue and object instance parameter should, for example, be `/= null`. If an error is detected the operator will cause an SDL dynamic error that will be treated as any other dynamic error found in an SDL system.

NewQueue: -> Queue

The literal `NewQueue` is used as an operator with no parameters and returns a reference to a new empty queue. The data area used to represent the queue is taken from an avail stack maintained by the list processing sorts. Only if the avail stack is empty new dynamic memory is allocated.

Cardinal: Queue -> Integer

This operator takes a reference to a queue as parameter and returns the number of components in the queue.

DisposeQueue: Queue -> Queue

This operator take a reference to a queue as parameter and returns all object instances and the data area used to represent the queue to the avail stack mentioned in the presentation of `NewQueue`. `DisposeQueue` always returns the value `null`.

> **Note:**
>
> Any references to an object instance or to a queue that is returned to the avail stack is now invalid and any use of such a reference is erroneous and has an unpredictable result.

Empty: Queue -> Boolean

This operator takes a reference to a queue as parameter and returns `false` if the queue contains any object instances. Otherwise the operator returns `true`.

FirstInQueue: Queue -> ObjectInstance

This operator takes a reference to a queue as parameter and returns a reference to the first object instance in the queue. If the queue is empty, `null` is returned.

Follow: Queue, ObjectInstance, ObjectInstance -> Queue

Follow takes a reference to a queue and to two object instances and inserts the first object instance directly after the second object instance. It is assumed and checked that the second object instance is a member of the queue given as parameter, and that the first object instance is not a member of any queue prior to the call.

Note:

The operator `Member` is used to check that the second object instance is member of the queue.

IntoAsFirst: Queue, ObjectInstance -> Queue

This operator takes a reference to a queue and to an object instance and inserts the object instance as the first object in the queue. The queue given as parameter is returned as result from the operator. It is assumed and checked that the object instance is not a member of any queue prior to the call.

IntoAsLast: Queue, ObjectInstance -> Queue

This operator takes a reference to a queue and to an object instance and inserts the object instance as last object in the queue. The queue given as parameter is returned as result from the operator. It is assumed and checked that the object instance is not a member of any queue prior to the call.

LastInQueue: Queue -> ObjectInstance

This operator takes a reference to a queue as parameter and returns a reference to the last object instance in the queue. If the queue is empty, `null` is returned.

Member: Queue, ObjectInstance -> Boolean

This operator takes a reference to a queue and to an object instance and returns `true` if the object instance is member of the queue, otherwise it returns `false`.

Precede: Queue, ObjectInstance, ObjectInstance-> Queue

Precede takes a reference to a queue and to two object instances and inserts the first object instance directly before the second object instance. It is assumed and checked that the second object instance is a member of the queue given as parameter, and that the first object instance is not a member of any queue prior to the call.

Note:

The operator `Member` is used to check that the second object instance is member of the queue.

Predecessor: ObjectInstance -> ObjectInstance

This operator takes a reference to an object instance and returns a reference to the object instance immediately before the current object instance. If the object instance given as parameter is the first object in the queue, `null` is returned. It is assumed and checked that the object instance given as parameter is a member of a queue.

Remove: ObjectInstance -> ObjectInstance

Remove takes a reference to an object instance and removes it from the queue it is currently a member of. A reference to the object instance is returned as result from the operator. It is assumed and checked that the object instance given as parameter is a member of a queue.

Successor: ObjectInstance -> ObjectInstance

This operator takes a reference to an object instance and returns a reference to the object instance immediately after the current object instance. If the object instance given as parameter is the last object in the queue,

`null` is returned. It is assumed and checked that the object instance given as parameter is a member of a queue.

NewObject: -> ObjectInstance

The literal `NewObject` is used as an operator with no parameters and returns a reference to a new object instance, which is not member of any queue. The data area used to represent the object instance is taken from an avail stack maintained by the list processing sorts. Only if the avail stack is empty new dynamic memory is allocated.

DisposeObject: ObjectInstance -> ObjectInstance

This operator take a reference to an object instance as parameter and returns it to the avail stack mentioned above. `DisposeObject` always returns the value `null`.

> **Note:**
>
> Any references to an object instance that is returned to the available stack are now invalid and any use of such a reference is erroneous and has an unpredictable result.

Examples of Use

In this section a number of examples will be given to give some indications of how to use the list processing "package". The following sort definitions are assumed to be included in the system diagram:

```
/*#include 'list1.pr' */

NEWTYPE ObjectDescr /*#NAME 'ObjectDescr'*/
  STRUCT
    SysVar SysTypeObject;
    Number Integer;
    Name   Charstring;
ENDNEWTYPE;

/*#include 'list2.pr' */
```

Example 548: Creating a Queue ––––––––––––––––––––––––––––––

To create a new queue and insert two objects in the queue, so that the first object has `Number = 23` and `Name = 'xyz'` and the second object has `Number = 139` and `Name = 'Telelogic'`, you could use the following code (assuming appropriate variable declarations):

```
TASK
  Q := NewQueue,
  O1 := NewObject,
  O1!ref!Number := 23,
  O1!ref!Name := 'xyz',
  Q := IntoAsFirst(Q, O1),
  O1 := NewObject,
  O1!ref!Number := 139,
  O1!ref!Name := 'Telelogic',
  Q := IntoAsLast(Q, O1);
```

Example 549: Removing from Queue

To remove the last object instance from a queue, assuming the queue is not empty, you could use the following code:

```
TASK
  O1 := Remove(LastInQueue(Q));
```

Example 550: Looking in Queue

You may look at the component `Name` in the first object instance in the queue in the following way:

```
TASK
  O1 := FirstInQueue(Q),
  StringVar := O1!ref!Name;
```

or if the reference to `O1` is not going to be used any further

```
TASK
  StringVar := FirstInQueue(Q)!ref!Name;
```

Example 551: Searching in Queue

The result of the following algorithm is that `O1` will be a reference to the first object instance that has the value `IntVar` in the component `Number`. If no such object is found `O1` is assigned the value `null`.

```
TASK O1 := FirstInQueue(Q);
NextObject:
DECISION O1 /= null;
  (true) :
    DECISION O1!ref!Number /= IntVar;
      (true):
        TASK O1 := Successor(O1);
        JOIN NextObject;
      (false):
    ENDDECISION;
  (false):
ENDDECISION;
```

Example 552: Removing Duplicates from Queue

The algorithm below removes all duplicates from a queue (and returns them to the avail stack). A duplicate is here defined as an object instance with the same `Number` as a previous object in the queue.

```
TASK O1 := FirstInQueue(Q);
NextObject:
DECISION O1 /= null;
  (true) :
    TASK O2 := Successor(O1);
    NextTry:
    DECISION O2 /= null;
      (true):
        DECISION O1!ref!Number = O2!ref!Number;
          (true):
            TASK Temp := O2,
                 O2 := Successor(O2),
                 Temp := DisposeObject (
                           Remove(Temp));
          (false):
            TASK O2 := Successor(O2);
        ENDDECISION;
        JOIN NextTry;
      (false):
        TASK O1 := Successor(O1);
        JOIN NextObject;
    ENDDECISION;
  (false) :
ENDDECISION;
```

Connection to the Monitor

Trace printouts are available for the operators in this abstract data type. By assigning a trace value greater or equal to eight (8) using the monitor command Set-Trace, each call to an operator in this data type causes a printout of the name of the current operator. Note that some of the operators may call some other operator to perform its task.

You may use the monitor command Examine-Variable to examine the values stored in a variable of type `ObjectInstance`. By typing an additional index number after the variable `Queue` the value of the `ObjectInstance` at that position of the queue is printed.

Accessing List Operators from C

The sorts `Queue`, `ObjectInstance`, and `ObjectDescr`, and all the operators and the literals `NewQueue` and `NewObject` have the same name in C as in SDL, as `#NAME` directives are used. The literal `null` is the sort `Queue` and is translated to `QueueNull()`, while the literal `null` in sort `ObjectInstance` is translated to `ObjectInstanceNull()`.

In C you access a component in an `ObjectInstance` using the -> operator:

```
OI_Var -> Component
```

As an example of an algorithm in C, consider the algorithm in Example 551 on page 3252. A reference to the first object instance that has the value `IntVar` in the component `Number` is computed:

```
#(O1) = FirstInQueue(#(Q));
while ( #(O1) != ObjectInstanceNull () ) {
  if ( #(O1)->Number == #(IntVar) ) break;
  #(O1) = Successor(#(O1));
}
```

Abstract Data Type for Byte

In this section an abstract data type for byte, i.e. `unsigned char` in C, is presented. This ADT can be used also in OS integrations and with Cmicro. However, please see the note below.

> **Note:**
>
> This ADT is only provided for backward compatibility, as the new predefined data type Octet should be used instead of Byte.

Purpose

The purpose of this data type is of course to have the type byte and the byte operations available directly in SDL.

The data type becomes available by including the file containing the definition with an analyzer included in an appropriate text symbol.

Example 553: ––

```
   /*#include 'byte.pr' */
 ––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
```

Available Operators

The following operators are available in this data type:

BAND: byte, byte -> byte

Bitwise and. Corresponds to C operator &

BOR: byte, byte -> byte

Bitwise or. Corresponds to C operator |

BXOR: byte, byte -> byte

Bitwise exclusive or. Corresponds to C operator ^

BNOT: byte -> byte

Unary not. Corresponds to C operator ~

BSHL: byte, integer -> byte

Left shift of the byte parameter the number of steps specified by the integer parameter. Corresponds to C operator <<

Implementation:

```
(byte)( (b << i) & 0xFF )
```

BSHR: byte, integer -> byte

Right shift of the byte parameter the number of steps specified by the integer parameter. Corresponds to C operator >>

```
Implementation: ( b >> i)
```

BPLUS: byte, byte -> byte

Byte plus (modulus 0xFF). Corresponds to C operator +

BSUB: byte, byte -> byte

Byte minus (modulus 0xFF). Corresponds to C operator -

BMUL: byte, byte -> byte

Byte multiplication (modulus 0xFF). Corresponds to C operator *

BDIV: byte, byte -> byte

Byte division. Corresponds to C operator /

BMOD: byte, byte -> byte

Byte modulus. Corresponds to C operator %

BHEX: charstring -> byte

This operator transforms a charstring (’00’ - ’ff’ or ’FF’) into a byte. The string may be prefixed with an optional ’0x’.

I2B: integer -> byte

I2B transforms an integer in range 0 - 255 into a byte.

B2I: byte -> integer

B2I transforms a byte into an integer.

Unsigned (and Similar) Types

There are three files called:

```
unsigned.pr
unsigned_long.pr
longint.pr
```

where three SDL sorts implemented in C as unsigned, unsigned long, and long int may be found. All these types are in SDL implemented as syntypes of integer. For more information please see the definitions of the data types.

Note:

These ADTs are only provided for backward compatibility, as is recommended to use the types in the package `ctypes` instead. The package `ctypes` is discussed first in this chapter.

How to Obtain PId Literals

This section describes a way to obtain `PId` literals for static process instances. `PId` literals will make it possible to simplify the start-up phase of an SDL system, as direct communication (*OUTPUT TO*) may be used from the very beginning. It is otherwise necessary to start sending signals without *TO*, as the only `PId` values known at the beginning are the *Parent* - *Offspring* relations.

Note:

This ADT **cannot** be used in OS integrations or with Cmicro. There are, however, a special version for OS integrations that can be found in the directory for the OS integration, and a special version for Cmicro that can be found in the Cmicro installation directory.

Note:

`PId` literals cannot be created for processes within block types or system types.

Purpose

In SDL the only way to obtain a `PId` value is to use one of the basic functions *Self*, *Parent*, *Offspring*, or *Sender*. Such values may then, of course, be passed as parameters in signals, in procedure calls and in create operations.

During system start-up there is no way to obtain the `PId` value for a static process instance at the output that starts a communication session. The receiver of the first signal must therefore be implicit, by using an output without *TO*.

To be able to handle outputs without TO, in SDL-92 types and in separate generated units, complete knowledge about the structure of channels and signal routes must be known at run-time. The same knowledge is also necessary if we want to check that there is a path from the sender to the receiver in an output with TO. As the information needed about channels and signal routes requires substantial amounts of memory, it would be nice, in applications with severe memory requirements, to be able to optimize this.

To remove all information about channels and signal routes from a generated application means two things:

1. Output without TO cannot be used in SDL-92 types or in separate generated units.
2. It is not possible to check that there is a path between the sender and the receiver at an output with TO.

The second limitation is no problem as this is the way we probably want it in a running application (during debugging the test ought to be used, but not in the application).

The first limitation, that output without TO cannot be used, is however more difficult. In an SDL system not using the OO concepts (block type, process type, and so on) and not using separate generation there are no problems, but otherwise such outputs are necessary at the system start-up phase to establish communication between processes in different blocks. The purpose of this abstract data type is to provide a way to establish `PId` literals and thereby to be able to avoid outputs without TO.

The Data Type PIdLit

Caution!

The PIdLit data type should only be used in the way described here to introduce synonyms referring to static process instances. Other usage may not work!

If you are using this data type in a system that is to be validated using the SDL Explorer there are two additional requirements:

- Only process types with the number of instances equal to (N,N) for N>0, may be referenced in PId_Lit operators.
- No process type with the number of instances equal to (N,N) for N>0, may contain a Stop symbol, independently if a PId_Lit operator is used for the process type or not.

The data type PIdLit contains the following operators:

```
PId_Lit : xPrsIdNode -> PId;
PId_Lit : xPrsIdNode -> PIdList;
PId_Lit : xPrsIdNode, Integer -> PId;
```

In the file containing the data type (`pidlist.pr`) there is also a synonym that you may use to access the environment:

```
SYNONYM EnvPId PId = ...;
```

The type `xPrsIdNode` corresponds to the C type `xPrsIdNode`, which is used to refer to the process nodes in the symbol table tree built up by a generated application.

Use the **first** version of `PId_Lit` to obtain a synonym referring to the process instance of a process instance set with one initial instance.

Use the **second** version of `PId_Lit` to obtain a synonym of array type referring to the process instances of a process instance set with several initial instances.

Use the **third** version of `PId_Lit` to obtain a synonym referring to one of the process instances of a process instance set with several initial instances.

To introduce PId literals implemented as SDL synonyms, follow the steps below:

1. Include the file pidlist.pr, which contains the implementation of the PIdList type, among the declarations in the system:

   ```
   /*#include 'pidlist.pr' */
   ```

2. Identify which process instance sets that should have PId literals.

3. Introduce #NAME directives for these process instance sets.

4. Insert a #CODE directive among the declarations in the system. If. however, separate generation is not used, this #CODE directive need **not** be included.

   ```
   /*#CODE
   #HEADING
   extern XCONST struct xPrsIdStruct
          yPrsR_ProcessName1;
   extern XCONST struct xPrsIdStruct
          yPrsR_ProcessName2;
   extern XCONST struct xPrsIdStruct
          yPrsR_ProcessName3;
   */
   ```

   There should be an external definition for each process instance set identified in step 2. ProcessNameX should be replaced by the name introduced in the #NAME directives for the processes.

5. For each process instance set that should have PId literals, introduce the following synonym definition in the system diagram.

   If the process type has **one initial instance**:

   ```
   SYNONYM Name1 PId =
     PId_Lit(#CODE('&yPrsR_ProcessName1'));
   ```

   If the process type has **several initial instances**:

   ```
   SYNONYM Name2 PIdList =
     PId_Lit(#CODE('&yPrsR_ProcessName2'));
   ```

   If the process type has **several initial instances, but only one of them should be possible to refer to by a synonym**:

   ```
   SYNONYM Name3 PId =
     PId_Lit(#CODE('&yPrsR_ProcessName3'), No);
   ```

where No should be the instance number, that is, if No is 2, then the synonym Name3 should refer to the second instance of the process type.

Of course, you may choose the names of the synonyms, but the string in the #CODE directive should be the xPrsIdNode variables in the extern definitions discussed in step 4 above.

6. You may now use the synonyms of type PId that you defined in step 5 in expressions of PId type, for example as a receiver in the TO clause in an output. The synonym EnvPId, which refers to an environment process instance, can be used in the same way.

   Synonyms of type PIdList may be indexed (as an array) by an integer expression to obtain a PId value and may then be used in the same way as the synonyms of type PId. Indexes should be in the range 1 to the number of initial instances.

Example 554: PIdList Data Type ----------

```
OUTPUT Sig1 TO Name1;
OUTPUT Sig2 TO Name2(2);
OUTPUT Sig3 TO Name2(InstNo);
OUTPUT Sig4 TO EnvPId;
DECISION (Name3 = Sender);
TASK PId_Variable := Name2(1);
```

where InstNo is an integer variable or synonym and PId_Variable is a variable of type PId.

Note:

Note that no index check will be performed when indexing a PIdList synonym.

General Purpose Operators

Introduction

The abstract data type `IdNode` described in this section introduces a number of operators that may be used to simplify an SDL system. The simplifications will give both reduced code size and higher speed of execution for your application, as well as make debugging easier. This ADT **cannot** be used in OS integrations or with Cmicro.

The operators may be grouped into two groups:

- "Almost SDL operations", that is, operators that are easy to understand in an SDL context, but which are not available in SDL. Examples are the possibility to enumerate all active instances of a certain process instance set, or to count the number of signals in an input port.
- Operators that handle implementation aspects. An example is an operator to reuse memory in avail lists.

Caution!

Be very careful using these operators, as you will then not be designing true SDL systems.

If the SDL description is a goal in itself you should not use the operators. If the SDL system is just a means to obtain something else, an application for example, the operators may be very useful.

Type IdNode

This abstract data type becomes available by inserting the analyzer include:

```
/*#include 'idnode.pr'*/
```

This abstract data type file introduces three SDL sorts called `PrsIdNode, PrdIdNode,` and `SignalIdNode` in SDL. These sort correspond to the types `xPrsIdNode, xPrdIdNode,` and `xSignalIdNode` in C, which are used to represent the symbol table in the generated application. The symbol table, which is a tree, will contain the static information about the SDL system during the execution of the generated program.

Is is possible to refer to processes, procedures, and signals (among others) using the following the names:

```
yPrsN_ProcessName     or &yPrsR_ProcessName
yPrdN_ProcedureName   or &yPrdR_ProcedureName
ySigN_SignalName      or &ySigR_SignalName
```

where *ProcessName*, *ProcedureName*, and *SignalName* should be replaced by the name of the process, procedure, or signal with prefix, or by the name given to the unit in a `#NAME` directive. To obtain a name of a unit with prefix the directive `#SDL` may be used:

```
yPrsN_#(ProcessName)   or &yPrsR_#(ProcessName)
```

To avoid problems when separate generation is to be used, the `&yPrsR_...` syntax is recommended.

The `#SDL` directive is not always possible to use. It will look for an entity with the specified name in the current scope unit (where the `#SDL` directive is used) and outwards in the scope hierarchy. So, for example, if the reference for a process is to be used in a process defined in another block, a `#SDL` directive cannot be used for the referenced process. The name of the referenced process ought then to be given in a `#NAME` directive.

If separate generation is used there may be more problems to access these references. The variables will be defined in the compilation unit where the entity they represent is defined.

- The `xPrsIdNode` for a process will be defined in the file containing the code for the block enclosing the process.
- The `xPrdIdNode` for a procedure will be defined in the file containing code for the enclosing unit.
- The `xSignalIdNode` for a signal will be defined in the file containing code for the enclosing system, block, or process.

A reference is visible in the compilation unit (file) where it is defined and in all subunits to the unit, as a compilation unit will include the `.h` file of all its parent units.

Problems occur when we want to use a reference in a place where it is not visible, for example using an `xPrsIdNode` for a process defined in a separate block, in a process in another block. All references are, however, extern, which makes it possible for a user to introduce an appro-

priate `extern` definition (in a `#CODE` directive) himself in the compilation units where it is needed.

Example 555 ----------

```
/*#CODE
#HEADING
extern XCONST struct xPrsIdStruct yPrsR_ProcessName;
*/
```

To know the name of the referenced process, a `#NAME` directive ought to be used.

> **Note:**
> Such `extern` definitions introduce dependencies between otherwise independent compilation units. It is your responsibility completely to maintain these dependencies.

Available Operators

GetIdNode: PId -> PrsIdNode;

This operator takes a `PId` value and returns a reference to the `PrsIdNode` that represents the process type. `PrsIdNode` values are not useful for anything except as parameters to the operators discussed here.

Kill: PId -> PId;

The `Kill` operator can be used to stop another process instance. In SDL a process instance may only stop itself. This operator has exactly the same effect as if the process instance given as parameter executed a stop operation. The `Kill` operator always returns the value `null`.

KillAll: PrsIdNode -> Integer;

This operator takes a reference to an `PrsIdNode` representing a process type and will kill all the instances of the specified process type. The effect is the same as if all the instances executed stop operations. The operator returns the number of “killed” process instances.

FirstPId: PrsdNode -> PId;

See “SucPId: PId -> PId;” on page 3264 (next).

SucPId: PId -> PId;

This operator, together with FirstPId, are intended to be used to enumerate all process instances of the process type referenced by the `PrsIdNode` given as parameter to `FirstPId`. `FirstPId` should be given a reference to an `PrsIdNode` for a process type and returns the first (last created) process instance. `SucPId` should be given a `PId` value and will return the next `PId` for the given process type.

> **Note:**
>
> During the enumeration of the process instances, no action that stops any instance of the enumerated process type may be executed.
>
> This means, for example, that the complete enumeration should take place in one transition and that `Kill` operations should not be used in the enumeration.

InputPortLength: PId -> Integer;

This operator returns the number of signals in the input port of the given process instance.

InputPortLength: PId, SignalIdNode -> Integer;

This operator returns the number of signals, of the signal type given as `IdNode` parameter, that are present in the input port of the given process instance. The `SignalIdNode` parameter should refer to a `SignalIdNode` that represents a signal or a timer.

NoOfProcesses: PrsIdNode -> Integer;

This operator should be given a reference to an `PrsIdNode` representing a process instance set and will return the number of active instances of this instance set.

IsStopped: PId -> Boolean;

The operator may be used to determine if a `PId` value refers to a process instance that is active or has executed a stop operation.

Broadcast: PrsIdNode, SignalIdNode, PId -> Integer;

This operator may be used to send one signal (without parameters) to each active process instance of a specified process instance set. The value of the third parameter, of type `PId`, will be used as sender in the signals. The result of the operator is the number of signals that are sent dur-

ing this operation, i.e. the number of active process instances of the specified type.

> **Note:**
>
> When you use this operator you hide signal sending in an expression in, for example, a task. This will decrease the readability of your SDL description, and should be well documented, at least with a comment.

FreeAvailList: PrsIdNode -> Integer and PrdIdNode -> Integer and SignalIdNode -> Integer

> **Note:**
>
> The FreeAvailList operator has no meaning in the SDL Explorer. It can be used but will in the Explorer be a null action.

The operator takes a reference to an `IdNode` (of one of the three type above) that represents a process, a procedure, or a signal and returns the memory in the avail list for the specified `IdNode` to the free memory by calling the `sctOS` function `xFree`. The function `xFree` uses the C standard function 'free' to release the memory. The `FreeAvailList` operator requires thus that free really releases the memory in such a way that it can be reused in subsequent memory allocations. Otherwise the operator is meaningless.

`FreeAvailList` is intended to be applied for reusing memory allocated for processes, procedures, and signals used only during a start-up phase. If the system, for example, contains a process used only during start-up, that is, all instances of this process perform stop actions early during the execution and no more processes will be created later, then the memory for these instances can be reused.

> **Caution!**
>
> This operator should only be used as one of the last resorts in the process of minimizing the memory requirements of an application.

Connection to Monitor

In the trace output, operators like `Kill` and `Broadcast` will produce trace messages exactly in the same way as the equivalent `Stop` operation and the sequence of `Output` operations.

Summary of Restrictions

The table below summarizes the restrictions concerning the usability of the various Abstract Data Types that are delivered with the SDL Suite.

	Sim.	Real-Time Sim.	Perf. Sim.	Sim. with env.	Appl. with Cadv.	Valid.
`list1, list2`	✓	✓	✓	✓	✓	✝
`file`	✓	✓	✓	✓	☛	☛
`random`	✓	✓	✓	✓	✓	☛
`pidlist`	✓	✓	✓	✓	✓	☛
`idnode`	✓	✓	✓	✓	✓	✓
`byte`	✓	✓	✓	✓	✓	✓
`longint`	✓	✓	✓	✓	✓	✓
`unsigned`	✓	✓	✓	✓	✓	✓
`unsigned_long`	✓	✓	✓	✓	✓	✓

Table Legend:

✓ Compatible

✝ Incompatible

☛ Meaningless combination, or restrictions. See the respective section for more information.

Chapter

63 *The Performance Library*

This chapter presents an overview of a method for performance simulation projects using the Performance Library. The Performance Library is available as an optional product.

The method is in no way complete or described in all details. The intention is not to describe the method itself in depth, but rather to present an application area for the SDL Suite. For more details about performance simulations in general, please refer to the literature in that field.

A Performance Simulation Project

To develop a performance simulation and use it to obtain estimates about a particular system involves a number of steps. The SDL Suite may be helpful in many, but not all, of these steps.

The major activities in a performance simulation project are:

1. Collect information about the behavior of the system to be simulated and define the purpose of the simulation, that is, what estimates should be the result of the project?

2. Create a performance model for the system. This is often expressed as a queuing network. As the behavior of a system is usually too complex to be simulated in all details, simplifications have to be made.

   This modeling phase is the most critical phase in the complete project. A good performance model leads to relevant results, while a bad model leads to nonsense. The key question is what simplifications and abstractions you may make and still obtain relevant results from simulations of the model.

3. Describe the performance model in detail in SDL and simulate it, using the library *Simulation*, until the SDL model behaves satisfactorily.

4. Execute the generated simulation a number of times to collect statistical data from the model.

   You will normally need 10,000 to 50,000 samples (of for example a queue length or a waiting time) to obtain any significance in the data. This means that program executions will be fairly long. There are two ways to execute the performance simulation:

   – Use the library *Simulation* and start by performing the monitor commands “Set-Trace 0" and “Go”, or

   – Use the library *PerformanceSimulation*. It is specially designed to execute performance simulations and does not include the monitor system. It will execute the performance simulation in the order **10 to 15** times faster than executing it using the library *Simulation*.

5. During the execution of the performance simulation, the best way to handle data measured in the system is to write the data on file. You may then analyze the data files using packages or programs for statistical analysis to obtain, for example, mean value, variance and confidence intervals for these estimates. (The SDL Suite does not provide means for statistical analysis.)

6. To validate these estimates it is common practice to compare the simulation results with results from mathematical methods applied on the system (simplified versions of the system). Queuing theory and the theory for queuing networks are the most relevant mathematical methods for such activities.

Some of these steps will discussed in more detail in the following subsections. For other aspects please see literature about performance simulations.

The Performance Model

Let us first state that a description of the functional behavior of a system and a performance model of the same system are (usually) not the same, even if both models can be expressed in SDL. The two models describe two different aspects of the same system. This is why there is a modeling phase in the development of the performance simulation.

Queuing Models

Most performance models can be viewed as queuing models or queuing networks, where a queuing network is an interconnected system of queuing models. A typical model for a queue would look like:

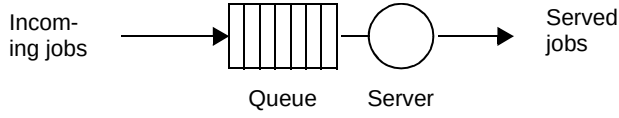


Figure 564: A queue model

The model contains jobs that need some service from the server. The jobs may have to wait in a queue for the service. The basic properties of this model are:

- The service time for a job. This can be modeled as a property of the job or of the server, depending on the system to be simulated.
- The queue discipline, that is, the order in which jobs are inserted into the queue and removed from the queue to get service. Typical disciplines are, to mention a few:
  - FCFS (First Come - First Served)
  - Priority order according to the priority of the jobs
  - Priority order with pre-emption.
- The inter-arrival time for jobs, or more generally: when new jobs are entered into the queue. The creation of new jobs is usually modeled in job-generators described as separate objects.

The queue model is a general model that may be used as an abstraction in many situations, for example to model programs (jobs) that are to be executed by the CPU in a computer (server). The queue is then the scheduled list of jobs that are in a “ready to execute” state. The queuing discipline is usually complex, involving for example priorities, pre-emption and cyclic execution of jobs.

Another quite different example would be a port, where ships (jobs) are coming for loading or unloading (the service). To perform loading or unloading the ship needs a crane (the server).

The systems that we want to simulate are usually not simple enough to be modeled by just one queue. However, models using interconnected queues-servers, connected in such a way that jobs leaving one server are inserted into the queue of another, have the power to describe many interesting real systems. An example of a simple queuing network model is given in Figure 565.

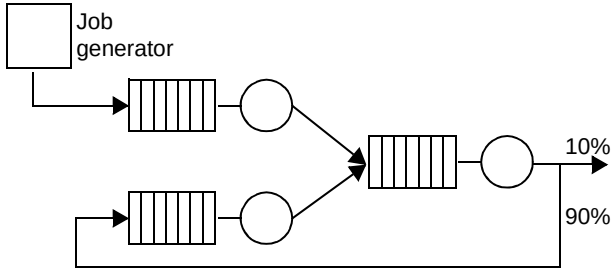


Figure 565. Simple queuing network

Each node in the network consists of one or several servers. The network also contains places where jobs are entered into the system and places where jobs are leaving the system.

Measurements

In a queue model or a queuing network model it is of interest to estimate for example:

- For jobs: total time in system, total waiting time
- For servers: load
- For queues: average and maximum queue length.

Such estimates can be obtained in two ways, using mathematical theories like queuing theory, or by measurements in simulations. With the mathematical theories, rather complicated models can be analyzed, usually more complicated than a nonspecialist thinks is possible. You should investigate this possibility before taking the decision to implement a simulation program.

Implementation of the Model

Mapping of Queue Models to SDL

A queuing network model may easily be described in SDL. Appropriate mapping rules are, for example:

- A server is implemented as a process and the queue as a variable in the server process.
- Jobs are described as data objects, that is, as passive entities that can be inserted into queues.
- Job generators are implemented as processes.
- Signals, with jobs as parameters, are used to send the jobs from job generators to servers and from servers to other servers.

Abstract Data Types for Queues and Random Numbers

In the implementation of the model there will be extensive use of queues and jobs and of queue manipulations, for example inserting a job into a queue. The chapter 62, *The ADT Library* provides an abstract data type specially designed for this purpose. See "Abstract Data Types for List Processing" on page 3243.

The ADT library also contains an abstract data type that can be used to generate random numbers. This data type can, for example, be used to draw job lengths and inter-arrival times according to a given distribution. See "Abstract Data Type for Random Numbers" on page 3234.

Random numbers are, in most situations in a performance simulation, used to model time intervals (for example service time required for a job) or to model a number of something (the number of jobs to be generated by a job generator at a certain time). The abstract data type for random numbers therefore contains the possibility to generate random numbers according to distributions returning non-negative values, for example:

- Negative exponential distribution
- Erlang distribution
- Hyperexponential distribution
- Uniform distribution
- Poisson distribution.

Implementation of Job Generators and Servers

To give a better understanding of job generators and servers, two process graphs are presented in Figure 566 and Figure 567, showing simple but typical processes.

A job generator sets a timer, and when the timer time expires, it generates a new job, sends it into the queuing system and sets the timer again. The time value in the set statements is usually Now + some random time interval. Note that the data types `Queue` and `ObjectInstance` are defined in the abstract data types for queue handling that are included in the ADT library (see "Abstract Data Types for List Processing" on page 3243 in chapter 62, *The ADT Library*).

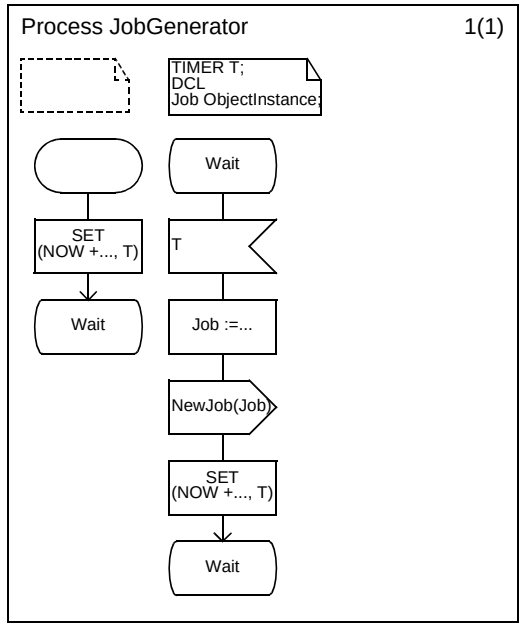


Figure 566: A job generator

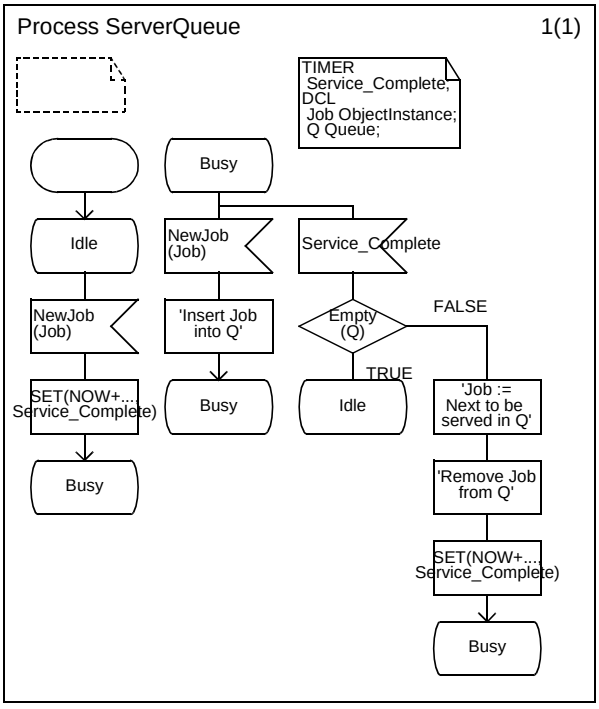


Figure 567: A server with a queue

The most important observation concerning the server process, is that the act of giving service is modeled by letting the server process wait in the state Busy. It is thus only the time needed for the service that is modeled. Otherwise the process is rather straight forward. The variable Job is used to refer to the job that is currently given service and the variable Q is used to store other jobs waiting for service. The queuing discipline will be implemented in the details in the tasks “Next to be served in Q”, “Insert Job in Q” and “Remove Job from Q”.

I/O and Performance Simulations

A performance simulation uses I/O mainly for two things:

1. Writing measurements on file
2. Prompting for parameters to the simulation.

A specially designed abstract data type for the purpose of simplifying I/O is included in the ADT library (see "Abstract Data Type for File Manipulations and I/O" on page 3221 in chapter 62, *The ADT Library*). With this data type it is possible to open files and to perform read and write operations in SDL.

It is, of course, interesting to parameterize a performance simulation on, for example, seed values for random generators, mean values for inter-arrival times and service times.

These values should be read at simulation start-up time. In SDL the concept of external synonyms can be used for such a purpose. As synonyms can only be defined in processes, not in instances, it is difficult to handle cases when the process instances should have different values for a certain simulation parameter. In such cases the abstract data type for I/O operations can be useful.

The second category of I/O mentioned above is printing of measurement data. There are basically two different types of measurements that have to be handled in a performance simulation.

1. Data concerning jobs such as waiting times.
   It is usually best to store this type of data in the job itself, until the job leaves the system, when the appropriate values are printed on a file.
2. Queue lengths and other related data.
   Such data is easiest to handle by introducing measurement processes that with regular time intervals print the queue lengths on file.

The best way to obtain the queue lengths is, in most situations, to let the measurement process view (or import) the appropriate queue variables. Otherwise a signal interface must be implemented only for measurement purposes.

The reason for printing all measurement data on file is that it is difficult to compute the relevant statistical entities at simulation time. If only the

mean values are of interest, these are simple to compute, but if variances and confidence intervals are to be computed it is better to let a professional statistical tool perform the job.

Note that data series produced from a simulation contains dependent data. If, for example, a job has been subject to a long waiting time, then the probability is high that the next job will also have a long waiting time.

Exit from a Simulation

The appropriate way to terminate the execution of a performance simulation is to call the function SDL_Halt(). It is best performed by introducing the call in a #CODE directive in a task symbol (see *"Including C Code in Task – Directive #CODE" on page 2728 in chapter 56, The Cadvanced/Cbasic SDL to C Compiler*). SDL_Halt is a function in the run-time library with the following prototype:

```
void SDL_Halt (void);
```

Execution with the Library Performance Simulation

As mentioned earlier, you may use the library *PerformanceSimulation* to speed up the execution of a performance simulation. This library has the same properties as the library *Simulation*, except that it does not contain the monitor system, which may increase the execution speed by a factor of 10 to 15.

Use the library *Simulation* to debug and verify the simulation model. Then select the library *PerformanceSimulation* in the make dialog in the Organizer (see "Make" on page 120 in chapter 2, *The Organizer*) and press the button *Make*. The thereby generated performance simulation may be executed from the OS as an ordinary program.

Chapter

64

Integration with Operating Systems

This chapter describes how to integrate code generated by the Cadvanced SDL to C Compiler with operating systems.

The different integration models (Light Integration, Threaded Integration and Tight Integration) are explained in the introduction.

There is a separate annex for each supported RTOS where the OS specific parts are described.

Important!

You are free to reuse the integrations supplied by IBM Rational or modify them to your needs. The OS integrations are tested by IBM Rational but IBM Rational does not guarantee that they will perform in your specific target environment (hardware, CPU, RTOS version etc.). Please refer to "RTOS integrations" on page 8 in chapter 1, *Platforms and Products, in the Installation Guide* for information about host and target environments where the integration types have been developed and tested.

Light and Threaded integrations

The Light and Threaded integrations are included in the delivery.

The standard product support and maintenance agreement **only** includes support for the Light and Threaded integrations available from IBM Rational if **no changes** have been made to the integrations.

Tight Integration

The Tight integration is meant to serve as a **template**, to be adapted to your needs. It is available for you as a free download from the IBM Rational Support web site.

The standard product support and maintenance agreement **does not** assist in adapting to your target environment.

Customer specific OS integrations

All OS integration models can be supported, enhanced and customized by using IBM Rational's Professional Services.

Other integrations

IBM Rational has developed a large number of integrations based on the company's vast experience of integrating with all operating systems on the market.

Introduction

The code that is generated by the Cadvanced SDL to C compiler is designed to run on different platforms. This is done by letting all platform dependent concepts be represented by C macros that can be expanded appropriately for each environment. There are also types used in the generated code that have to be defined. Integration, as referred to in this chapter, is the process of adapting the generated code to a certain platform.

This chapter describes the different models that are supported in SDL Suite.

> **Note:**
> Throughout this chapter, annexes excluded, VxWorks terminology has been used whenever there are differences between operating systems. Particularly, this means that the term *task* has been used on several occasions. The corresponding term would be *thread* for Win32 and Solaris, *process* for OSE Delta, and *task* for Nucleus.

Different Integration Models

With Cadvanced, there are three different run-time models, called Light Integration, Threaded Integration and Tight Integration. Tight integrations are then divided into two submodels, the Standard model and the Instance-Set model. All models use the same generated code.

You will find descriptions of the different models below, as well as guidelines for choosing between them.

Light integration

The simplest case is called a Light integration because only a minimum of interaction with the operating system is required; a Light integration could even run on a target system without any OS at all.

The complete SDL system runs as a single OS task. Scheduling of SDL processes is handled by the standard kernel, running a complete state transition at a time (no preemption). Worst case scheduling latency is thus the duration of the longest transition.

Communication between SDL processes is handled by the kernel; communication between the SDL system and the environment is handled in

the two user supplied functions `xInEnv`() and `xOutEnv`(). These functions are platform dependent. See Figure 568.

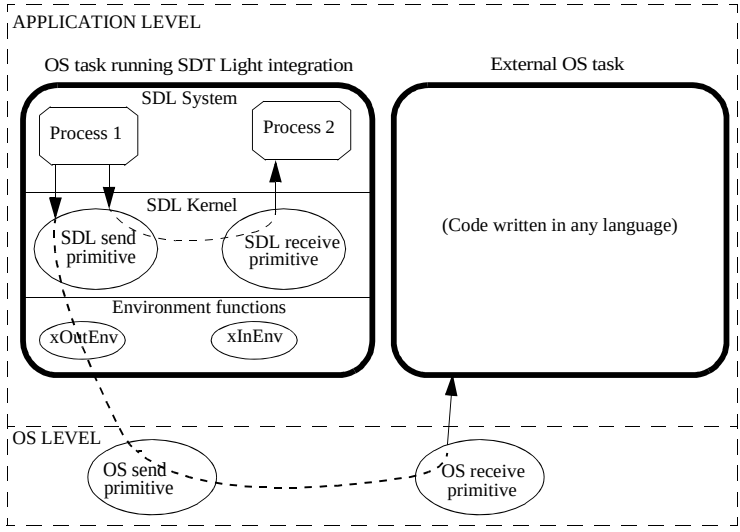


Figure 568: Signalling in the Light Integration Model.

An SDL process sends one internal and one external signal. `xOutEnv`*() must be written to call the OS send primitive. Thick borders denote OS tasks.*

Other properties of a Light Integration:

- The scheduling order can be controlled by process priorities and/or signal priorities as set by the `#PRIO` directive.
- An SDL system can be partitioned, i.e. split into several executables. Each partition has its own kernel/scheduler and set of environment functions. Partitioning is explained in *"Partitioning" on page 2641 in chapter 56, The Cadvanced/Cbasic SDL to C Compiler*.
- It is easy to "go to target": Write the environment functions and recompile the standard kernel with a cross compiler.

Threaded Integration

The main difference between a Light integration and a Threaded integration is that any part of the SDL system can execute in its own thread in a Threaded integration. A thread in a Threaded integration can exe-

cute one or several SDL processes or even blocks. How the different SDL objects should be mapped to threads is specified in the Deployment Editor. The user can specify thread-specific parameters like: STACKSIZE, PRIORITY, MAXMESSAGEQUEUESIZE and MAXMESSAGESIZE in the Deployment Editor. The Deployment Editor works together with the Targeting Expert to generate a Threaded integration.

Note:

A Threaded integration can only be generated using the Deployment Editor and the Targeting Expert, i.e. the make feature in the Organizer CANNOT be used.

Communication and execution control in one thread is handled by the SDL kernel and not the OS kernel. See Figure 569. This model shows the default Threaded integration where OS semaphores are the only OS primitives used in the signal sending. The `xMainLoop()` function is the entry point for each thread.

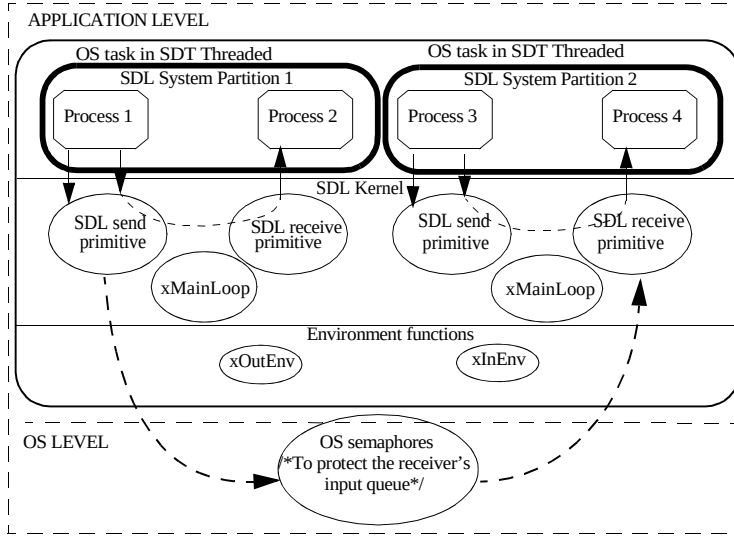


Figure 569: Signalling in Threaded Integration.

SDL Kernel functions are used for sending signals within or between threads. Thick borders denote threads.

Semaphores are used frequently in Threaded to protect globally accessible data like input queues and export queues. Semaphores are also used in the start-up to synchronize the execution of newly created threads. We must ensure that no thread is allowed to start executing until ALL threads have been created in the start-up phase.

Communication with external threads (Threads not made from SDL specifications) is handled by the environment functions in the same way as for a Light integration. There is though one major difference and that is that it is possible in a Threaded to send signals directly to an SDL process with the `SDL_Output()` function. The `SDL_Output()` function is "thread-safe" because of the use of OS semaphores.

Variations of Threaded Integration

There are two different implementation of signal sending between threads in Threaded. The default model send signals in the same way as in a Light integration but the input queues are protected by OS semaphores. In the alternative model, the signals are sent/received by OS_Send and OS_Received. In the alternative model there is no use for OS semaphores. Which model to use can be decided at compilation time by setting the compilation switch `THREADED_ALTERNATIVE_SIGNAL_SENDING`.

Threaded Default

The sender of an SDL signal "takes" a semaphore before accessing the receivers input queue. The signal is then linked into the input queue and the semaphore is "given" back.

The receiver is normally "waiting" for a semaphore. When a signal is sent, the semaphore is "released" by the sender. To send a signal, two semaphores are normally needed. One semaphore is protecting the input queue and the other is used for synchronizing the sender and the receiver. In Solaris, where we use POSIX threads and semaphores they are called: xInputPortMutex and xInputPortCond.

One OS feature that is needed in Threaded is a "Conditional Wait". We are only allowed to wait for a signal until the first internal timer expires. In POSIX, there is a primitive called `pthread_cond_wait()` and in Windows there is a similar concept called `WaitForSingleObject()` where a time-out can be specified. In VxWorks and OSE, the only concept where you can specify a time-out is OS_Receive. For synchroniza-

tion between the sender and receiver we are sending a small OS_Message (the character ‘c’) in these two RTOS.

Threaded using OS Send and OS Receive

In this alternative implementation we are sending the signals using OS_Send and OS_Receive. Signals are now sent to OS_Message queues. The sender sends the pointer to the SDL signal in an OS_Message to the receiver’s OS_Queue. When the signal is received, the SDL signal is unpacked and linked in to the receivers input queue. The advantage with this implementation is that there is now synchronization between the sender and the receiver. Any number of threads can send signals at the same time to a specific receiver without having to wait for a semaphore.

Tight integration

We also provide an alternate run-time model, which is called a Tight integration because the generated code interacts directly with the underlying operating system when creating processes, sending signals, etc.

The SDL processes run as separate OS tasks as explained below. Scheduling is handled by the OS and is normally time-sliced, priority based and preemptive.

Communication takes place using the inter-process communications mechanisms offered by the OS, normally message queues. This applies to signals sent between SDL processes as well as signals sent to or received from the environment. There are no environment functions, as illustrated in Figure 570.

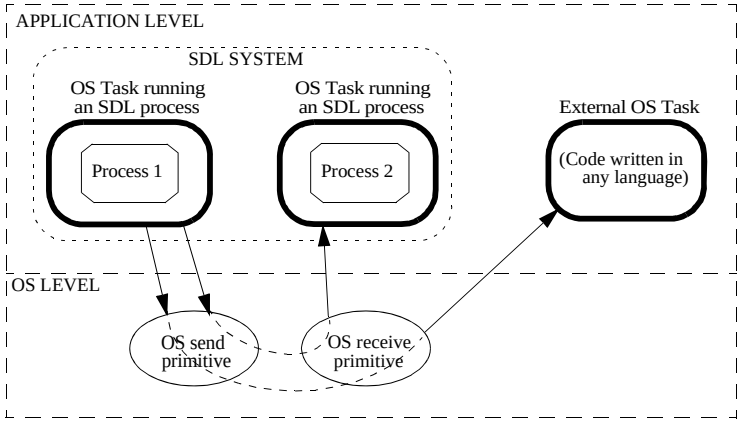


Figure 570: Signalling in the Tight Integration Model.

An SDL process sends one internal and one external signal. OS primitives are always used. The SDL system is not an entity of its own. Thick borders denote OS tasks.

Other properties of a Tight Integration:

- There is a single timer task which handles all SDL timers.
- SDL Simulators cannot be tightly integrated with an RTOS.
- Execution trace is available in textual or MSC format.[1]

Variations on Tight integration

Tight integrations come in two varieties, the Standard model and the Instance-Set model. Consider an SDL system with processes as outlined in Figure 571.

1. The MSC trace will be printed on standard output. To see the trace in an MSC diagram, copy the text into a file and read it into the MSC Editor.

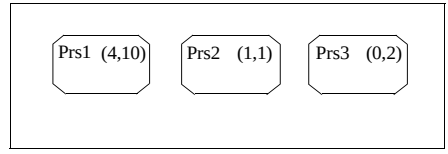


Figure 571: A partial SDL system

This system will be used for explaining the different models of Tight Integration.

Tight Standard

In the standard model of a Tight integration every SDL process instance is mapped to its own OS task. Tasks are created and destroyed dynamically as SDL process instances are created and terminated.

The example system will initially run as five OS tasks (four for Prs1, and one for Prs2). Up to 13 tasks may be created as needed.

Tight Instance-Set

In the Instance-Set model of a Tight integration every SDL process instance set is mapped to its own OS task. Tasks are created statically when the SDL system is initialized and are never destroyed. Thus there may be OS tasks that have no SDL process instances to run.

The example system will run as three OS tasks regardless of the number of process instances (one task each for Prs1, Prs2 and Prs3).

Choosing between Light, Threaded and Tight Integration

Choosing between the three integration models depends on many factors. Some of them are related to properties of the SDL system, others relate to the target environment. Important factors to consider are:

- The trade-off between performance and scheduling latency.
- How complex interaction the system has with the environment.
- Whether an operating system will be used or not.
- Memory management

Performance vs. scheduling latency

Generally, the Light integration model provides better performance than a Tight integration, but worst-case scheduling latency is the duration of the longest state transition.

A Tight integration is normally preferable when low scheduling latency is important. On the other hand, performance suffers from the overhead associated with OS scheduling and inter-process communications.

Another consideration is that blocking calls (either inlined in SDL code or as part of environment functions) will completely stop execution of a Light integration for the duration of the call. Making blocking calls in a Tight integration will only stop the thread making the call.

The Threaded integration is in a sense a combination of the other two integration models. An application part that makes blocking calls can be mapped to a separate OS thread. Other parts, where fast inter-process communication is important, can be mapped to another OS thread. Internally in this thread, signal exchange will be handled in the same way as in a Light integration.

Environment interaction

If interaction with the environment is simple then a Light integration is the best choice, especially if the system sends many signals and receives few. You can generate templates for the environment functions from the SDL Suite and just add code for converting between the signal representation and the actual environment hardware or software.

If interaction with the environment is complex then a Tight integration is probably the easiest to use, since you only need to interface to the operating system queues. This is the case if you need process behavior in the interaction (for example, to establish a communications session before sending the signal).

Other cases where a Tight integration might be the best choice are

- when the environment consists of many OS tasks written in other languages than SDL
- when external processes send signals directly to SDL processes rather than to the SDL system in general
- when there are many signals passing to and from the environment

In a Threaded integration, the integration with the environment should normally be handled in the environment functions in the same way as for a Light integration. It is possible though in Threaded to send signals directly to an SDL process with the `SDL_Output()` function without involving the `xInEnv()` function.

Operating system issues

If you will not use an OS at all then you have to select a Light integration. In addition, you will have to provide some simple functions for getting system time and handling memory allocation (depending on compiler and libraries, standard C library functions can often be used).

If you use an OS that takes care of load balancing between CPUs then you should select a Tight integration, because load balancing normally uses threads as the load unit to distribute.

If you want to distribute your SDL System over several nodes (CPUs) you should use the Threaded integration together with the TCP/IP feature.

Considerations when choosing between Tight and Threaded

Overall, IBM Rational recommends using the Threaded integration mode, unless there are specific technical reasons to prefer one of the other integration models.

The two most important reasons for choosing the Threaded model are:

1. The Threaded integration out-perform a Tight Integration in most situations, e g implementation of Timers and creation of tasks is about 100% faster in Threaded.
2. In Threaded you are not limited to two partitioning models. Any mapping between SDL objects and threads can be specified in the Deployment editor.

For a more comprehensive list of the differences between Threaded and Tight see the following sections.

Advantages of Threaded

- The default model is very simple to implement for a new OS. A working prototype should normally take 2-3 days.

- The Threaded model can be used together with the TCP/IP feature.
- The Threaded model is supporting all partitioning models (1 process instance/1 or many process instance-set/1 or many blocks/1 entire system mapped to one Thread.
- The user can easily specify Thread specific parameters like: stack size, Thread priority, Thread message queue size and maximum message size for each Thread.
- The Thread specific code for all supported OS is placed in one file.

Advantages of Tight

- Simple mapping od SDL concepts to OS primitives.
- Simple interactions with environment (at least in the default model), by direct call to the OS message passing primitives.
- Support textual and MSC-pr textual trace in console window when executing.

Disadvantages of Threaded

- Slightly more difficult to interact with the environment. An external function/thread must use the SDL_Output function when sending signals to an SDL process.

Disadvantages of Tight

- Very difficult and time consuming to support a new OS.
- Can only support two partitioning models.
- Very inefficient Timer model.

Common Features

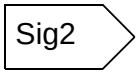
This section describes the parts of the integration that are common to the different models, but also some important differences.

> **Note:**
> Many of the data structures and macros described in this section are described in more detail in chapter 61, *The Master Library*. That chapter is, however, focused on a Light Integration. Some things, especially the listings of data structures, are not correct in every detail for Tight Integrations but should still be very useful.

The Use of Macros

A source file generated by an SDL to C compiler is independent from the choice of integration model and operating system. Instead of system calls, it uses macros that have to be defined elsewhere when the system is built. Each SDL concept is represented by one or more C macros. These macros will have different definitions for different integration models and operating systems. The SDL Suite provides a number of integration packages for this purpose. In a Light Integration, many macros are expanded into functions of the standard kernel. A Tight Integration has lower level macros that are defined in separate files for each operating system, and finally expanded into OS primitives or certain OS dependent constructions.

Below is an example of generated code for signal sending. Code in capital letters are the SDL Suite macros. The bracketed numbers indicate corresponding lines of code. For a description of the different macros, see chapter 61, *The Master Library*.



SDL symbol: Output

Generated code before macro expansion:

```
[1]   ALLOC_SIGNAL(sig2, ySigN_z3_sig2,
        TO_PROCESS(Env, &yEnvR_env),
        XSIGNALHEADERTYPE)
      SIGNAL_ALLOC_ERROR
[2]   SDL_2OUTPUT_COMPUTED_TO(
        xDefaultPrioSignal,
        (xIdNode *)0, sig2,
        ySigN_z3_sig2,
        TO_PROCESS(Env, &yEnvR_env),
        0, "Sig2")
      SIGNAL_ALLOC_ERROR_END
```

Generated code after macro expansion for a Light Integration:

```
[1]   yOutputSignal = xGetSignal
        ((&ySigR_z3_sig2),
        (*(&yEnvR_env)->
        ActivePrsList !=
        (xPrsNode) 0 ?
        (*(&yEnvR_env)->
        ActivePrsList)->Self :
        xSysD.SDL_NULL_Var),
        yVarP->Self);
[2]   SDL_Output (yOutputSignal,
        (xIdNode *) 0);
```

File Structure

> **Note:**
>
> The source file and examples for Tight Integration are not included in the standard delivery. They are available as free downloads from IBM Rational Support web site.

The Generated Files

An integration uses one or more of four files that can be generated by the Code Generator. All integrations require the C source file `<systemname>.c`, whereas the interface file `<systemname>.ifc` is optional. For a Tight Integration the signal number file `<systemname>.hs` may be used, and for a Light Integration the file `sctenv.c`, representing the environment, can be used.

The source file uses the highest level set of macros, that are defined in the different integration packages.

The Integration Packages

The files containing the necessary macro definitions and support functions are organized as shown in Figure 572. For each operating system there is one package for the Light Integration, and one for the Tight Integration. Although the files in different packages may have the same name they do not necessarily contain the same code. The principles for Tight Integration packages are described more thoroughly in "Tight Integration" on page 3340. Details about each operating system can be found in the annexes.

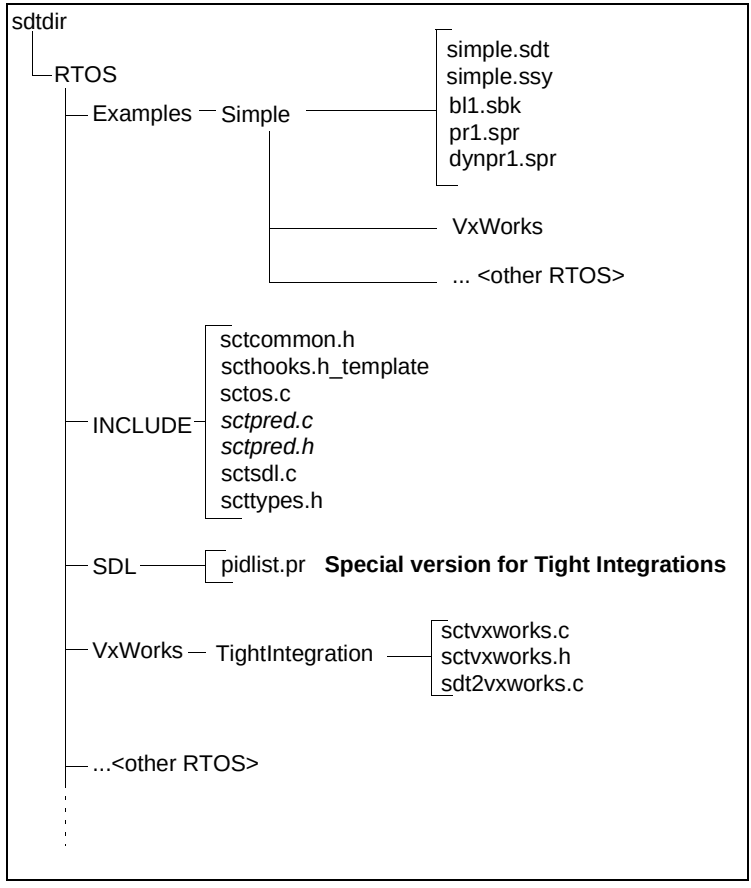


Figure 572: File structure for the RTOS integrations

There is also an INCLUDE directory containing source files common to all supported operating systems. These files are described in "File Structure" on page 3025 in chapter 61, *The Master Library*. Below are some additional comments on these files:

- scthooks.h:

  This file contains macro hooks into the system. These hooks are used for customizing a Tight Integration, for example by adding external tasks. See the examples to find out how this is done.

  All the hook macros are initially empty.

- sctcommon.h:

  This file contains general macros for all Tight Integrations

- sctos.c:

  This file contains functions that are operating system and/or compiler dependent, like allocation and free of dynamic memory.

- sctpred.h and sctpred.c:

  This is the header and source file where all the SDL predefined datatypes are implemented.

- sctsdl.c:

  This file contains SDL support functions, the timer task and the timer support functions. It is non-OS-specific and calls many second level OS-specific macros defined in sct<RTOS>.h.

- scttypes.h:

  This file contains the general datatype definitions for signals, IdNodes, etc. It also contains the macro definitions found in generated code. Note that this file is non-OS-specific. This means that if a call to an OS-specific primitive is needed, then a second level of macro is defined, according to the following model.

  In the generated code:

  ```
  ALLOC_SIGNAL_PAR(ok, ySigN_z3_ok, TO_PROCESS(Env,
  &yEnvR_env), yPDef_z3_ok)
  ```

  In scttypes.h:

```
#define ALLOC_SIGNAL_PAR(SIG_NAME, SIG_IDNODE, \
   RECEIVER, SIG_PAR_TYPE) \
   RTOSALLOC_SIGNAL_PAR(SIG_NAME, SIG_IDNODE, \
   RECEIVER, SIG_PAR_TYPE)
```

In `sct<RTOS>.h`:

```
#define RTOSALLOC_SIGNAL_PAR(SIG_NAME,SIG_IDNODE,\
   RECEIVER, SIG_PAR_TYPE) \
   yOutputSignalPtr = \
    xAlloc(sizeof(xSignalHeaderRec)+ \
    sizeof(SIG_PAR_TYPE)); \
   yOutputSignalPtr->SigP = yOutputSignalPtr+1; \
   yOutputSignalPtr->SignalCode = SIG_NAME; \
   yOutputSignalPtr->Sender = SDL_SELF;
```

The `Examples` directory contains a simple SDL system that also uses an external process (a separate OS task). For each supported operating system there is an implementation of this, demonstrating how to hook into the integration package. Further description of the example can be found in "A Simple Example" on page 3355.

Naming Conventions

Names of variables, datatypes and support functions in generated code and package files often start with one of the letters x, y and z.

The general rules (there are some exceptions) are:

- Names and objects starting with an 'x' represent general datatypes and support functions in the kernel.

  Examples: `extern XCONST struct xVarIdStruct, xInputSignal, xFindReceiver`

- Names starting with a 'y' are names of IdNodes representing SDL variables, process states, channels, blocks, datatypes for signals, PAD functions, etc. in generated code.

  Examples: `extern XCONST struct xVarIdStruct yVarR_z012_okmess, ySigR_z3_ok, yPAD_z01_pr1`

- Names and objects starting with a 'z' are SDL variables, SDL names, process state names, etc. in generated code.

  Examples: `#define z010_idle 1, z012_okmess`

The Symbol Table

All signals, blocks, processes, channels, etc. in an SDL system have a corresponding representation in the symbol table described in chapter 61, *The Master Library*. This symbol table consists of nodes (`IdNodes`) each representing one entity of the system. The `IdNodes` are pointers to structs. See the following example:

```
extern XCONST struct xPrsIdStruct yPrsR_z02_dynpr1;
#define yPrsN_z02_dynpr1  (&yPrsR_z02_dynpr1)
```

The `N_` and the `R_` just before `z02_dynpr1` indicate if it is a node (`N_`) or a record (`R_`).

Memory Allocation

The `xAlloc` function is always used when allocating dynamic memory. The function is placed in the `sctos.c` file, but in the Tight Integrations the body of the function is found in the `sct<RTOS>.h` file.

Start-up

The `yInit` function is called during start-up of the SDL system. It is responsible for creating all static processes and for initializing SDL synonyms.

Implementation of SDL Concepts

SDL Processes

An SDL process consists of three parts in generated code: Instance set common data, instance specific data and dynamic behavior.

Instance set common data

Variables and structures that are common to all instances of a process are stored in a record of the type `xPrsIdStruct`, defined in `scttypes.h`. This record is referenced by a node in the symbol table.

Instance specific data

Variables and structures of the process instance are declared via the macro `PROCESS_VARS`. This macro is defined in different ways in the Light and the two models of Tight Integration. It contains state informa-

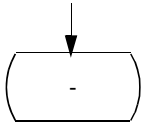
tion, local variables, pointers to parent and offspring, etc. One important entry is the `RestartAddress`, pointing out which transition to execute when the PAD function runs (see below).

Dynamic behavior

The dynamic behavior of an SDL process is implemented in a PAD function (Process Activity Definition). The PAD function is used somewhat differently in the different integration models:

- In the Light Integration the scheduler calls the PAD function of the process. The PAD function then returns to the scheduler when the transition is finished.
- In the Standard Model Tight Integration the PAD function is called when the process is started and does not return until process termination. It then contains a main loop where one iteration corresponds to one transition.
- The Instance Set Model Tight Integration contains a mix of the two.

Below is the code for ending a transition, before and after macro expansion for a Light Integration.



SDL symbol: Nextstate
Generated code before macro expansion:

```
      /*-----
      * NEXTSTATE -
      * #SDTREF(SDL,/ti/RTOS/MANUAL/SDL/simple.spr(1),
                143(55,100),1)
      ------*/
      #ifdef XCASELABELS
[1]     case 6:
      #endif
[2]     XAT_LAST_SYMBOL
[3]     SDL_DASH_NEXTSTATE
```

Generated code after macro expansion for a Light Integration:

```
      /*-----
      * NEXTSTATE -
      * #SDTREF(SDL,/ti/RTOS/MANUAL/SDL/simple.spr(1),
                143(55,100),1)
      ------*/

[1]     case 6:

[2]       xGRSetSymbol (-1);
[3]       SDL_NextState (VarP, yVarP->State);
          return;
```

Signal Queues

Basically, every process has at least one signal queue.

- In a Light Integration each process instance has an input queue.
- In the Standard Model Tight Integration there is an input queue and a save queue for each instance.
- In the Instance Set Model Tight Integration there is an input queue and a save queue for every instance set. These two queues are shared by all the instances.

All signals that are sent to a process arrive in the input queue. The save queue is used to keep signals that cannot be handled in the current state but should be saved for future use. This is tightly connected to the SDL Save concept, but is also used in the implementation of timers. For a Light Integration there are no save queues. Instead, all signals that should be saved will remain in the input queue until they can be received.

Process Priorities

The use of process priorities requires some caution. Priorities can be set with the `#PRIO` directive in the SDL Suite, but there is no mapping of priorities for different platforms. The generated code will use exactly the values specified in the SDL system. This will not be a problem in a Light Integration, but for Tight Integrations the result may not be the expected.

Example 556: Process priority problems ––––––––––––––––––––––

Assume that Process1 has its priority set to 100 using the `#PRIO 100` directive and Process2 has `#PRIO 50`. These priority values will be used as-is by the underlying scheduler.

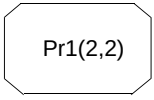
In the SDL Suite simulators and Light integrations, the highest priority is 0. In VxWorks, 0 is the highest priority, whereas in pSOS the highest priority is 255.

Thus the simulated system, a Light integration for any operating system and a Tight integration for VxWorks will all run Process2 at a higher priority than Process1. In contrast, a Tight integration for pSOS will run Process1 at a higher priority than Process2!

––

Process Creation

Regardless of the integration model there are a number of things that have to be done when an SDL process instance is created. The structs that represent the instance have to be created. It needs a representation in the symbol tree and a signal queue, except in the Instance Set Model of Tight Integration where the signal queue belongs to the instance set. A start-up signal is also always allocated and sent to the process.



SDL symbol: SDL Static Create

Generated code before macro expansion:

```
      /*********************************************************
                    SECTION      Initialization
      *********************************************************/

      extern void yInit XPP((void))
      {
        int  Temp;
      ........
        INIT_PROCESS_TYPE(pr1,z01_pr1,yPrsN_z01_pr1,"z01_pr1",
                          SDL_INTEGER_LIT(2),SDL_INTEGER_LIT(2),
                          yVDef_z01_pr1, xDefaultPrioProcess,
                          yPAD_z01_pr1)
      #ifdef SDL_STATIC_CREATE
[1]     for (Temp=1; Temp<=SDL_INTEGER_LIT(2); Temp++) {
[2]        SDL_STATIC_CREATE(pr1,z01_pr1,yPrsN_z01_pr1,
                             "pr1",ySigN_z01_pr1,
                             yPDef_z01_pr1,yVDef_z01_pr1,
                             xDefaultPrioProcess,yPAD_z01_pr1,1)
        }
      #endif
```

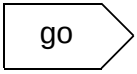
Generated Code after macro expansion for a Light Integration:

```
      /*********************************************************
                       SECTION      Initialization
      *********************************************************/
      extern void
      yInit ()
      {
        int  Temp;
      ..........
[1]     for (Temp = 1; Temp <= 2; Temp++) {

[2]       SDL_Create (xGetSignal ((&ySigR_z01_pr1),
                      xSysD.SDL_NULL_Var, xSysD.SDL_NULL_Var),
                     (&yPrsR_z01_pr1), 1);
        }
```

SDL Signals

Signal Sending

In generated code a signal sending is handled by two macros: `ALLOC_SIGNAL` (or `ALLOC_SIGNAL_PAR` for a signal with parameters) and `SDL_2OUTPUT_xxxx` (there are different macros depending on how the SDL output was defined, e.g. with or without an explicit TO).



SDL symbol: Output

Generated code before macro expansion:

```
[1]   ALLOC_SIGNAL(go,ySigN_z03_go,TO_PROCESS(p,yPrsN_z09_p),
                   XSIGNALHEADERTYPE)
      SIGNAL_ALLOC_ERROR
[2]   SDL_2OUTPUT_COMPUTED_TO(xDefaultPrioSignal,(xIdNode*)0, go,
                ySigN_z03_go,TO_PROCESS(p, yPrsN_z09_p), 0, "Go")
      SIGNAL_ALLOC_ERROR_END
      XBETWEEN_SYMBOLS(4, 579)
```

Generated code after macro expansion for a VxWorks Tight Integration:

```
[1]   yOutputSignalPtr = xAlloc(sizeof(xSignalHeaderRec));
[1]   yOutputSignalPtr->SignalCode = 2;
[1]   yOutputSignalPtr->Sender = yVarP->Self;
[2]   Err = msgQSend (xTo_Process ((&yPrsR_z09_p)),
[2]                   (char*)*&yOutputSignalPtr,
[2]                   sizeof(xSignalHeaderRec) + 0, 0, 0);
[2]   xFree ((void **) &yOutputSignalPtr);
[2]   if (Err == (-1)) {
[2]     taskLock ();
[2]     printf ("Error during %s found in VXWORKS
[2]     function %s. Error code %s\n", "OUTPUT",
[2]     "msgQSend", strerror ((*__errno ())));
[2]     taskUnlock ();
[2]   }
```

In this example the signal is called go and has no parameters. The `SDL_2OUTPUT_COMPUTED_TO` macro indicates that it was sent without an explicit TO.

SDL Procedures

An SDL procedure is represented by a function similar to a PAD function. Before a procedure is called there are two support functions that need to be called: `xGetPrd` and `xAddPrdCall`.

The `xGetPrd` function allocates an `xPrdStruct` for the called procedure and returns an `xPrdNode` pointing to the struct.

The `xAddPrdCall` function adds the new procedure call in the calling process' `ActivePrd` list (an element in the `xPrsStruct`).

The procedure is called with a pointer to the instance data of the calling SDL process. This is because the procedure must be able to use internal variables in the calling process.

Before a procedure returns to the caller it performs an `xReleasePrd` call. This function removes the call from the `ActivePrd` list.

SDL Timers

SDL timers are represented by signals. All active timer signals are kept in a sorted list, either within the single task of a Light Integration or in a certain timer task in a Tight Integration. When a timer expires, the signal representing it is sent to the SDL process that set it.

SET
(Now+5,T1)

SDL symbol: SET Timer

Generated code before macro expansion:

```
      /*-----
      * SET T1
      * #SDTREF(SDL,/ti/RTOS/MANUAL/SDL/dynpr1.spr(1),
                119(55,25),1)
      ------*/
      #ifdef XCASELABELS
[1]     case 2:
      #endif
[2]       SDL_SET_DUR(xPlus_SDL_Time(SDL_NOW,
[3]        SDL_DURATION_LIT(5.0, 5, 0)),
[4]        SDL_DURATION_LIT(5.0, 5, 0), t1, ySigN_z021_t1,
             yTim_t1, "T1")
```

Generated code after macro expansion for a Light Integration:

```
      /*-----
      * SET T1
      * #SDTREF(SDL,/ti/RTOS/MANUAL/SDL/dynpr1.spr(1),
                119(55,25),1)
      ------*/

[1]     case 2:

[2]       SDL_Set (xPlus_SDL_Duration (SDL_Now (),
[3]         SDL_Duration_Lit (5, 0)), xGetSignal
[4]         ((&ySigR_z021_t1), yVarP->Self,
            yVarP->Self));
```

Light Integration

A Light Integration is a stand-alone executable which can be generated with or without a simulator. An executable that should run under UNIX can use the precompiled kernels. Only the environment functions need to be written by the user.

PAD Functions

The PAD function is called by the scheduler when its process is in turn to execute a transition. The scheduler calls the PAD function with a symbol table node of the type `xPrsNode`, pointing to the instance specific data of the instance that is scheduled.

Start-Up

A Light Integration starts when the generated main function is called. The start-up phase works like this (pseudocode shown in *italics*):

```
void main( void )
{
   xMainInit();
   Code from #MAIN
   xMainLoop();
}


void xMainInit( void )
{
   xInitEnv();
   Init of internal structures
}
```

You must supply the `xInitEnv()` function to initialize external code and hardware, etc. (this is of course application dependent). This function is placed in the same program module (environment module) as the `xInEnv()` and `xOutEnv()` functions. The `xMainLoop()` function contains an eternal loop, which constitutes the scheduler itself. See below:

```
void xMainLoop (void)
{
   while (1)
   {
     xInEnv(...)
     if (a timer has expired)
       send the corresponding timer signal
     else if (a process can execute a transition)
      {
       remove the signal from the input port
       set up Sender in the process to Sender of the signal
       call the PAD function for the process
      }
   }
}
```

Connection to the Environment

Signals going in and out of the SDL system are handled in the two user written functions `xInEnv` and `xOutEnv`. There is a template file for writing these two function in the standard distribution. This file can be found at `<installation directory>/sdt/sdtdir/<your platform os version>sdtdir/INCLUDE/sctenv.c`.

Running a Light Integration under an External RTOS

Since there are some fundamental differences between different RTOS we can only give a general idea of how to generate a Light Integration under an external RTOS here. Typical things that may be different in different RTOS:

- If you are allowed to have a main function in your application.
- If your start-up function must be specified in a configuration file.
- If the cross compiler requires additional OS-specific header files to be included.
- If it is possible to run the application in a simulated target environment.
- Syntax for the makefile.

General Steps

The normal steps to create a Light Integration under an external RTOS can be summarized as follows:

1. Copy the source and header files for an application kernel from the installation of the SDL Suite. The files are residing in the following directory:

   `<installation directory>/sdt/sdtdir/<your host and os version>sdtdir/INCLUDE`.

2. Generate an `<application>.c` file with the SDL to C Compiler.

Note:

The code generator option *Cadvanced* **must** be used.

3. Generate an environment header file (an option in the Organizer *Make* dialog).
4. Edit the `sctenv.c` template file to handle your in and out environment signals. Include the generated `<application>.ifc` file (the environment header file).
5. Create a makefile or edit the generated makefile. Write entries for the kernel source files, the environment file and the application file.
6. Set the appropriate compilation switches for your RTOS and your compiler.
7. Compile the application and the kernel file to create a relocatable object file.
8. Download.

Threaded Integration

Introduction

The first part of this section is about the Threaded integration. The second part is about the TCP/IP feature. With the Threaded and TCP/IP features you can partition an SDL system into several threads that can execute in a distributed environment.

The "New threaded integrations" on page 3326 described last has been developed by IBM Rational in the IBM Rational Tau product. Our intention, however, is that the integrations should be possible to use both in SDL Suite (Cadvanced) and Tau/Developer (AgileC and C Code Generator) products.

Implementation Details for Threaded

The main areas where a Threaded differs from a Light integration are:

- Symbol table structures and global variables
- Process creation
- Signal sending

Symbol Table Structures and Global Variables

Each thread has a global variable of type `xSystemData` in Threaded. In Light, there is only one global variable of this type for the entire system. In Threaded this variable is initialized in `yInit()` and is one of the parameters in the macro for creating a new thread.

The data structure `xSystemData` has a couple of new entries:

```
...
#ifdef THREADED
   xInputPortRec  xNewSignals;
   THREADED_THREAD_VARS
#endif
...
```

The entry `xNewSignals` is used when a new signal is received. The signal is first linked into the receiver's `xNewSignals` queue before it is handled in the receiver's `xMainLoop()`.

The entry THREADED_THREAD_VARS is a macro that contains different thread variables like the semaphore handles xInputPort.

Global variables are declared in the macro THREADED_GLOBAL_VARS:

```
#elif THREADSOLARIS
.....
#define THREADED_GLOBAL_VARS \
  pthread_mutex_t xListMutex; \
  pthread_mutex_t xExportMutex; \
  sem_t xInitSem; \
  int xNumberOfThreads; \
  int QueueCounter; \
  int xInitPhase;  \
  pthread_attr_t Attributes ;
#endif /* THREADED_ALTERNATIVE_SIGNAL_SENDING */
#endif
```

Process Creation

There are three macros related to the creation of threads in a Threaded:

- THREADED_START_THREAD(F, SYSD, THREAD_STACKSIZE, THREAD_PRIO, THREAD_MAXQUEUESIZE, THREAD_MAXMESSIZE)

  This macro is used if there is only one thread to be created for all instances (Instance Set) of an SDL process (or another SDL object). The parameters are:

Parameters	Explanation
F	The entry point of the thread, i.e. the SDL kernel function xMainLoop()
SYSD	A variable of type xSystemData
THREAD_STACKSIZE	The stacksize for the thread is specified in the Deployment Editor
THREAD_PRIO	The thread priority specified in the Deployment Editor

Parameters	Explanation
THREAD_MAXQUEUESIZE	The maximum number of messages/signals in the input queue of the thread, as specified in the Deployment Editor. This parameter is only used if the alternate signal sending model is used.
THREAD_MAXMESSIZE	The maximum size of a message/signal. This parameter is ignored at the moment in both implementation models. The maximum size of a message/signal in the alternate signal sending model is always the size of the pointer to an SDL signal (sizeof(xSignalNode).

Note:

Thread parameters for individual threads can be specified in the Deployment Editor. If no values are specified, default values will be used.

- SDL_CREATE(PROC_NAME, PROC_IDNODE, PROC_NAME_STRING)

  This macro is used for creation of dynamic processes (processes created during run-time).

  It is defined exactly the same way in Threaded as in Light. It will call the SDL kernel function SDL_Create().

  SDL_Create() will create a new thread if, for instance, there should be one thread for each instance of an SDL process. See the following extract from the SDL_Create() function:

```
.....................
#ifdef THREADED
 if (PrsId->SysD == 0) {
    THREADED_START_THREAD(xMainLoop, StartUpSig-
>Receiver.LocalPId->PrsP->SysD, PrsId->ThreadParam-
>ThreadStackSize, PrsId->ThreadParam->ThreadPrio,
PrsId->ThreadParam->MaxQueueLength, PrsId-
>ThreadParam->MaxMesSize);
}
#endif
```

```
.....................
```

Please note that the thread parameters are taken from the `xPrsIdNode` for the process.

```
SDL_STATIC_CREATE(PROC_NAME, PREFIX_PROC_NAME,
PROC_IDNODE, PROC_NAME_STRING, STARTUP_IDNODE,
STARTUP_PAR_TYPE, PRIV_DATA_TYPE, PRIO,
PAD_FUNCTION, BLOCK_INST_NUMBER)
```

This macro is used for creating static processes (processes that are created at system start-up). It will be called from the `yInit()` function and is mapped to the SDL kernel function `SDL_Create()` in the same way as in the `SDL_CREATE` macro.

The `xMainLoop()` function is the entry point for the thread. First in this function is the macro

```
THREADED_THREAD_BEGINNING(SYSD)
```

The main purpose of this macro is to wait for the start-up semaphore `xInitSem`. No thread is allowed to start executing until ALL static processes/threads have been created.

Sending Signals

In the default model, signals are sent in the same way as for a Light integration except that they are first linked into the `xNewSignals` queue.

In the `xMainLoop()` function, the `xNewSignals` queue is checked for new entries. If a new signal is available, it is sent to the process itself with the `SDL_Output()` function. See the following extract from the `xMainLoop()` function:

```
........
THREADED_LOCK_INPUTPORT(((xSystemData *)SysD))
while (((xSystemData *)SysD)->xNewSignals.Suc != (xSig-
nalNode)&((xSystemData *)SysD)->xNewSignals)
SDL_Output(((xSystemData *)SysD)->xNewSignals.Suc
xSigPrioPar(((xSystemData *)SysD)->xNewSignals.Suc-
>Prio), 0);
THREADED_UNLOCK_INPUTPORT(((xSystemData *)SysD))

.........
```

The Alternative Signal Sending Model.

In the alternative signal sending model, an OS message/signal is sent containing a pointer to the SDL signal. For this, a message queue must

be created for each thread at thread creation. How this is done differs from OS to OS. See : Signalling in Threaded Integration.

The sender sends the pointer to the SDL signal with the OS_Send primitive. The receiver receives the message, extracts the pointer to the SDL signal and links it into the xNewSignal queue. After this, the signal is handled in the same way as in the default model.

New Macros

The same source files are used in a Threaded integration as in a Light. The only SDL kernel source files that are affected by the Threaded integration are `scttypes.h`, `sctsdl.c` and `sctos.c`.

Most of the OS specific code is found in the `scttypes.h` file. The following macros are new for Threaded:

MACRO NAME	Explanation
`THREADWIN32`	Main macro for Threaded Windows integration
`THREADSOLARIS`	Main macro for Threaded Solaris integration
`THREADVXWORKS`	Main macro for Threaded VxWorks integration
`THRAEDOSE`	Main macro for Threaded OSE integration
`THREADED_ALTERNATIVE_SIGNAL_SENDING`	Main macro for using the alternative signal sending model.
`THREADED_POSIX_THREADS`	Main macro for the Threaded integration model defining the kernel specifics
`THREADED`	Internal macro used only in the kernel source files
`THREADED_TRACE`	Enabling textual execution trace
`THREADED_OSTRACE`	This macro is mapped to a printf statement.

MACRO NAME	Explanation
THREADED_ERROR	Enabling textual error printout when calling OS primitives.
THREADED_ERROR_VAR	Defines a variable used in THREADED_ERROR_RESULTS.
THREADED_ERROR_RESULT	Stores the return result after calling an OS primitive.
THREADED_ERROR_REPORT	Checks the value of the return variable and if ERROR makes a printout.
THREADED_ERROR_REPORT_NULL	Checks the value of the return variable and if ERROR makes a printout.
THREADED_ERROR_REPORT_NEG	Checks the value of the return variable and if ERROR makes a printout.
THREADED_ERROR_REPORT_OPEN_NEG	Checks the value of the return variable and if ERROR makes a printout.
THREADED_ERROR_REPORT_WAIT_NEG	Checks the value of the return variable and if ERROR makes a printout.
THREADED_GLOBAL_VARS	Global variable defines.
THREADED_GLOBAL_INIT	Initialization of global variables like semaphores.
THREADED_THREAD_VARS	Definitions of thread variables.
THREADED_THREAD_INIT	Initialization of thread variables.
THREADED_THREAD_BEGINNING	Wait for xInitSem to be released.
THREADED_LOCK_INPUTPORT	Protect the input queue by taking the semaphore.
THREADED_UNLOCK_INPUTPORT	Releasing the semaphore for the input queue.

MACRO NAME	Explanation
THREADED_WAIT_AND_UNLOCK_INPUTPORT	Wait for next message/signal to arrive or the next internal timer to expire.
THREADED_SIGNAL_AND_UNLOCK_INPUTPORT	Send a signal and release the semaphore for the input queue.
THREADED_LISTREAD_START	Protect global active and available lists with a semaphore before reading it.
THREADED_LISTWRITE_START	Protect global active and available lists with a semaphore before writing to it.
THREADED_LISTACCESS_END	Release the semaphore protecting a global active or available list.
THREADED_EXPORT_START	Protect export and import actions by taking a semaphore
THREADED_EXPORT_END	Release the semaphore after export and import actions.
THREADED_START_THREAD	Start a new thread.
THREADED_STOP_THREAD	Terminate a thread.
THREADED_AFTER_THREAD_START	Synchronize the start-up of newly created threads.
THREADED_SEND_OUTPUT	Send messages/signals. Used in both models.

Textual and MSC Trace in Threaded

Textual SDL Trace, similar to the Textual Trace in the SDL Suite Simulator, can be turned on by selecting the flag *SDL trace* (will set the flag THREADED_XTRACE) in the Target Library/Kernel window in Targeting Expert.

On-line MSC trace is possible when running an application under a soft-kernel on Windows or UNIX. Select the flag *MSC trace* (will set the flag THREADED_MSCTRACE) in the Target Library/Kernel window in Targeting Expert.

API for interfacing a Threaded Integration

An API is available with a number of useful functions/MACROS that will facilitate the work of sending/receiving signals between external processes/threads and an SDL Threaded Application.

The following functions and MACROS are available:

SDL_PId xThreadedRegExtTask()

This function will return an SDL PId representing the calling External process/thread. It works in the following way:

1. Get a ThreadId for the calling process/thread.
2. Allocate and assign an SDL struct (`xPrsIdRec`) representing the external process/thread.
3. Call the SDL kernel function `xGetPId()` to get an `SDL_PId`.
4. If MSC trace is on it will generate an entry for the process/thread in the MSC diagram.
5. Allocate and initialize an SDL "system data record" for the external process/thread.
6. Create a queue for the calling process/thread.
7. Assign the "system data record" entry in the `SDL_PId`.
8. Return the `SDL_PId`.

xSignalNode xThreadedReceiveSDLSig(SDL_PId)

This function will wait for an SDL signal indefinitely. When a signal arrives the signal will be taken out of the queue and return the signal. If MSC trace is on the signal will also be traced in the MSC diagram.

xSignalNode xThreadedReceiveSDLSig_WithTimeOut(SDL_PId, SDL_Duration)

This function will work in the same way as `xThreadedReceiveSDLSig()` except that it will only wait for the specified time.

THREADED_ASSIGN_SDL_SIG_PARAMS(sigptr,signame,paramno,param)

This macro will use the macro token operator ## to concatenate the sigptr, signame, paramno into a simple assignment of the signal parameter with the specified number.

> **Note:**
> This macro should only be used with simple datatypes where assignments can be done using the simple assignment operator "="

THREADED_GET_SDL_SIG_PARAM(sigptr,signame,paramno,param)

This macro will assign the user's param the value of the signal parameter with the specified number.

> **Note:**
> This macro should only be used with simple datatypes where assignments can be done using the simple assignment operator "="

SDL_Output()

This is the standard SDL kernel function that should be used when sending signal into a Threaded Application. The following must be done before a signal can be sent from an external process/thread into a Threaded Application.

1. Use the `xThreadedRegExtTask/xThreadedRegExtTask_WithQueue` function to get an `SDL_PId`.
2. Call the SDL kernel function `xGetSignal` with the following parameters: `signalId`, `Receiver`, `Sender`.
3. Assign signal parameters using the API macro `THREADED_ASSIGN_SDL_SIG_PARAMS()`
4. Call the SDL_Output function with the SDL signal.

For an example see Annex 6: Building a Threaded Integration

THREADED_START_EXTTASK

This macro is normally empty. It is called after all static SDL threads are created in the `THREADED_AFTER_THREAD_START` macro (last in generated c file for application).

By defining this macro to user can make the application start external tasks.

THREADED_SIMPLE_EXAMPLE

If this flag is defined the external tasks in the simple example will be started.

Implementation Details for Different RTOS

VxWorks

The Threaded integration for VxWorks is developed and tested using a Windows Softkernel.

The following VxWorks header files are used:

```
#ifdef THREADVXWORKS

#include "errno.h"
#include "vxWorks.h"
#include "semLib.h"
#include "msgQLib.h"        /* msgQCreate msgQDelete ms-
gQSend msgQReceive *//* msgQNumMsgs */
#include "taskLib.h"         /* taskSpawn */
#include "semaphore.h"       /* POSIX semaphores */
#endif
```

The following VxWorks primitives have been used:

VxWorks primitives	Explanation
`sem_init(..),` `sem_wait(..),` `sem_post(..),` `sem_destroy(..),`	POSIX semaphores are frequently used, e.g. to protect the input queue, to synchronize start-up...
`msgQCreate(..),` `msgQReceive(..),` `msgQSend(..),` `msgQDelete(..)`	Message queues are used in both models in VxWorks. The message queue is created in the `THREADED_START_THREAD` MACRO.

VxWorks primitives	Explanation
`taskSpawn(..), taskDelete(..), taskSuspend(..)`	taskSpawn is used for creating a thread. The name entry is not used. taskSuspend is only used by the "Main" thread. It is called from the macro `THREADED_AFTER_THREAD_START`.
`taskIdSelf()`	Used in taskDelete() and taskSuspend()

Thread Parameters	Default values
`DEFAULT_STACKSIZE`	800
`DEFAULT_PRIO`	100
`DEFAULT_MAXQUEUESIZE`	250
`DEFAULT_MAXMESSIZE`	sizeof(xSignalNode)

Solaris

The Threaded integration for Solaris is developed and tested in the following environment:

Sun Solaris 2.6 with Sun WorkShop compiler cc version 5.0.

Included header files for Solaris:

```
#elif THREADSOLARIS
#include <pthread.h>
#include <semaphore.h>
#ifdef THREADED_ALTERNATIVE_SIGNAL_SENDING
#include <mqueue.h>
#include <signal.h>
#include <time.h>
#endif /* THREADED_ALTERNATIVE_SIGNAL_SENDING */
#endif
```

The following Solaris primitives have been used:

Solaris primitives	Explanation
`sem_init(..),` `sem_wait(..),` `sem_post(..),` `pthread_mutex_init(..),` `pthread_mutex_lock(..),` `pthread_mutex_unlock(..),` `pthread_mutex_destroy(..)` `pthread_cond_init(..),` `pthread_cond_wait(..),` `pthread_cond_timedwait(..),` `pthread_cond_signal(..),` `pthread_cond_destroy(..)`	The sem_.. primitives are only used in the start-up to synchronize static processes/threads. The pthread_mutex_.. primitives are used to protect the input queues and other queues. The pthread_cond_... primitives are used to synchronize sender and receiver in the default model.
`pthread_attr_init(..),` `pthread_attr_setstacksize(..),` `pthread_attr_setschedpolicy(..),` `thread_attr_setdetachstate(..),` `pthread_attr_setscope(..)`	The pthread_attr_.. primitives are used to set the attributes of a thread before it is created.
`mq_open(..),` `mq_close(..),` `mq_receive(..),` `mq_unlink(..),` `mq_send(..)`	These primitive are only used in the alternative signal sending model.
`pthread_create(..),` `pthread_exit(..),`	These primitives are used when a thread is created and when it terminates.
`timer_settime(..),` `timer_delete(..)`	These primitives are used for setting a timer before the mq_receive() is called in the alternative signal sending model. The time-out for the timer is the duration for the next internal SDL timer to expire. When the timer expires the signal_handler function sets the error message to EINTR.

Thread Parameters	Default values
DEFAULT_STACKSIZE	15000
DEFAULT_PRIO	10
DEFAULT_MAXQUEUESIZE	128
DEFAULT_MAXMESSIZE	sizeof(xSignalNode)

OSE

The Threaded integration for OSE have been developed and tested with an OSE Softkernel (version 4.3) on Solaris 2.6 using the gcc compiler (version 2.95).

The following OSE header file is include:

```
#ifdef THREADOSE
#include "ose.h"
#endif /* THREADOSE */
```

The following OSE primitives have been used:

OSE Primitives	Explanation
create_sem(..), wait_sem(..), signal_sem(..),	These primitives are used for protecting input queues, other queues and for synchronizing start-up
receive(..), receive_w_tmo(..), send(..)	These primitive are used for receiving/sending signals in both models. Receive/send is also used to pass the start-up parameter xSysD to a new thread, since OSE does not support start-up parameters in the create_process primitive.
alloc(..), free_buf(..)	These primitives are used for allocating and returning OSE signals.

OSE Primitives	Explanation
`start(..)`	This primitive is used for starting a newly created thread.
`kill_proc(..)`	This primitive terminates a thread
`stop(..)`	This primitive stop the execution of a thread. Used in the `THREADED_AFTER_THREAD_START` macro.
`current_process()`	Used in the kill_proc() and stop() calls.
`create_process(..)`	This primitive is used for creating Threads.

Thread Parameters	Default values
`DEFAULT_STACKSIZE`	1024
`DEFAULT_PRIO`	8
`DEFAULT_MAXQUEUESIZE`	1024
`DEFAULT_MAXMESSIZE`	sizeof(xSignalNode)

Declaration of xMainLoop().

The declaration and definition of the thread's entry point (xMain-Loop()) is special in OSE:

Declaration in scttypes.h;

```
.......
#elif THREADOSE
extern OSENTRYPOINT xMainLoop;
#else
extern void xMainLoop XPP((xSystemData *));
#endif
```

Definition in sctsdl.c:

```
......
#elif THREADOSE
```

```
OS_PROCESS(xMainLoop)
#else
#ifndef XNOPROTO
void xMainLoop (xSystemData * SysD)
#else
void xMainLoop (SysD)
                                xSystemData * SysD;
#endif
#endif
..............
```

Definition of NIL.

NIL is defined in the OSE kernel and must not be redefined in the SDL kernel.

```
#ifndef THREADOSE
#define NIL 0
#endif /* THREADOSE */
```

Forward declaration of xSignalNode.

The xSignalNode must have forward declaration very early in the scttypes.h file since it is used in the union SIGNAL definition.

```
.............
typedef struct xSignalStruct *xSignalNode;
union SIGNAL
{
  SIGSELECT      sigNo;
  xSignalNode    SDLSig;
  xSystemDataPtr SysD;
};

.....
```

Further down in the scttypes.h file.

```
.....
#ifndef THREADOSE
typedef struct xSignalStruct       *xSignalNode;
#endif /* THREADOSE */
.....
```

Windows

The Threaded integration for Windows is developed and tested on Windows XP with the Microsoft C++ compiler.

The following header files for Windows are included:

```
#ifdef THREADWIN32
#include "limits.h"
#include "windows.h"
#include "dos.h"
......
```

The following Windows primitives are used in the Threaded integration:

Windows primitives	Explanation
`CreateSemaphore(..)`, `ReleaseSemaphore(..)`, `WaitForSingleObject(..)`, `CloseHandle(..)`	These primitives are used for protecting input queues, other queues and synchronization in start-up. One extra semaphore, xInitQueue, is used when a newly created thread is creating his own input queue.
`PeekMessage(..)`	This primitive is used in the `THREADED_THREAD_BEGINNING` macro in the alternative signal sending model. This primitive force the thread to create a message queue. It is used together with the xInitQueue semaphore.
`GetMessage(..)`, `PostThreadMessage(..)`,	These primitives are used for receiving/sending messages in the alternative signal sending model.
`SetTimer(..)`, `KillTimer(..)`	These primitives are used to set an OS timer that will signal the thread when it expires. The timeout of this timer is the duration until the next internal SDL timer expires for this thread. The window message will be set to `WM_TIMER` if the OS timer expires.

Windows primitives	Explanation
`CreateThread(..), ExitThread(..)`	These primitives are used for creating and terminating threads.
`SetThreadPriority(..)`	This primitive is used for setting the priority of the thread.
`SuspendThread(..)`	Called by the “main” thread in the macro `THREADED_AFTER_THREAD_START`
`GetCurrentThread()`	Used in the SuspendThread macro.

Thread Parameters	Default values
`DEFAULT_STACKSIZE`	0 (Automatically resized by OS)
`DEFAULT_PRIO`	`THREAD_PRIORITY_NORMAL`
`DEFAULT_MAXQUEUESIZE`	1024
`DEFAULT_MAXMESSIZE`	sizeof(xSignalNode)

Signal Sending over TCP/IP

Introduction

For applications using the Threaded integration model, a plug-in module for TCP/IP communication is available. The module supports signal sending between distributed SDL applications via a TCP/IP connection. ASCII Encoding/Decoding is used for the conversion between signal and transport format. The module is delivered as C source code which is integrated and built together with code generated by the CAdvanced SDL to C Compiler.

The TCP/IP adapter supports the four operating systems for which Threaded integrations are available. These are Windows, Solaris, VxWorks and OSE.

Architecture

The TCP/IP functions are called from the environment functions `xInitEnv` and `xOutEnv`. The functions are included by setting the `XENV_INC` flag to `"tcpipcomm.h"`. The `tcpipcomm.h` file contains `#define` directives that translates macros in the environment file to function calls in the TCP/IP adapter.

When a signal is sent to the environment using the `xOutEnv` function, `xSendSignal` is called. A connection is set up with a remote server. The signal destination is specified using a user-implemented function. When the connection is accepted, the signal is encoded into ASCII format and sent via a TCP/IP socket. The session is then closed.

From `xInitEnv`, a thread is started which polls a socket for incoming connections. When a connection from a remote client is accepted, a new thread is started, which receives and decodes data for one signal. When the signal is decoded, it is inserted in the signal queue of the SDL system. The thread finishes its execution after the connection is closed.

An executing SDL system thus acts as a server when signals are received from the environment and as a client when a signal should be sent to the environment.

The environment file of an SDL application may not be modified if the TCP/IP adapter is to be used. Making modifications may override the TCP/IP signal sending functionality. If you want to use the TCP/IP adapter together with other external code from an environment file, please consult "Configuration" on page 3324.

File Structure

The TCP/IP adapter consists of four files, located in `$sdtdir/tcpip/`.

- `tcpipcomm.c` should be compiled and linked with the generated C code
- `tcpipcomm.h` is a C header file which is included from the generated environment file
- `tcpipthr.h` and `tcpipsock.h` are header files that are included from `tcpipcomm.c` and `tcpipcomm.h`.

The tcpip directory is referenced relative to the $sdtdir variable.

Hint: Using TCP/IP with a new $sdtdir

If you use an $sdtdir that is different from the default $sdtdir (in the SDL Suite installation directory), be sure to copy the **tcpip** directory to the new location. Otherwise, you may encounter problems in finding the TCP/IP files at compilation.

Note: Pointers as Signal Parameters

Distributed components execute in separate memory spaces. Care must be taken so that pointers are not sent as signal parameters over TCP/IP and used in a remote component.

Routing of Signals

For each signal that is sent from your SDL application, you must specify a destination in the form of an IP address (or host name) and a TCP port number. This information should be accessed from a routing function which is called when a signal is sent from the SDL application.

The routing function is made accessible from the TCP/IP adapter by setting the flag XROUTING_INC to the name of a routing header file. This is a plain C header file where the macro XFINDDEST(OUTSIG, SIGNAME, IP, PORT) is defined as a function. OUTSIG and SIGNAME are **in** parameters. IP and PORT are **out** parameters.

OUTSIG should be declared as xSignalNode*. From this parameter, signal data such as parameters can be accessed. SIGNAME holds the name of the signal and is declared as char*. In many cases, the signal name is sufficient routing information. The IP and PORT variables should be set to the host address and port number. IP should be declared as char* and PORT as int*.

The routing function should be implemented in a C file which is compiled and linked together with the application. The server IP address should be given as char*, e.g. "255.255.255.255" (dotted decimal notation) or "server.the_company.com" (hostname notation). The server port number should be given as int, e.g. 8888.

On the server side, the IP address is inherent to the host that the application resides on. The port number on which a server should listen for incoming connections should be specified using the flag XSERVPORT.

Note: Usage of Port Numbers

Generally, TCP port numbers below 1024 are reserved by operating system services or internet applications. For instance, the port numbers 21 and 23 are used by FTP and port number 80 is used by HTTP. If a port number is occupied, you will get an error message at start-up and the server thread will not start. It is recommended that you select port numbers larger than 1024 for your SDL application.

Example 557: TCP/IP Adapter Routing Settings ––––––––––––––––––

This example shows a situation where different signals should be directed to different recipients. An SDL system is partitioned into three components (i.e. executable files) using the Threaded integration model. From component 1, two different signals can be sent. Sig1 should be sent to component 2 and Sig2 should be sent to component 3.

Component 2 resides on a host called "host2". It uses port number 7001 for listening for incoming connections, which means that XSERVPORT is set to 7001 (XSERVPORT=7001). Component 3 resides on "host3" and listens on port number 7001 (XSERVPORT=7001). Please note that the ports are not in conflict since the hosts are different.

For component 1, a routing function is implemented. The flag XROUTING_INC is set to "router.h". The routing header file has the following contents:

```
#define XFINDDEST(OUTSIG, SIGNAME, IP, PORT)\
xGetDestination(OUTSIG, SIGNAME, IP, PORT)
```

The routing C file, router.c, contains the implemented function:

```
#include "stdlib.h"

void xGetDestination(xSignalNode *sig, char
*sigName, char *IPAddr, int *Port)
{
  if (strcmp(sigName, "Sig1") == 0)
  {
    strcpy(IPAddr, "host2\0");
    (*Port)= 7001; /* Dereferencing */
  }
  else if (strcmp(sigName, "Sig2") == 0)
  {
    strcpy(IPAddr, "host3\0");
```

```
      (*Port)= 7001;
    }

    return;
  }
```

`router.c` is then compiled and linked together with the generated code, the coder library and the TCP/IP adapter. This example shows a fairly trivial routing scenario. The routing is based only on the name of the signal. The xSignalNode pointer is not used.

Error Handling

The TCP/IP adapter contains basic error handling. Error checks are performed when encoding/decoding, socket and thread functions are invoked. If `THREADED_ERROR` is defined, an error message is printed on stdout with the name of the function where the error occurred. A platform-dependent error code is included in the error message. For a description of the error code, consult the User's Manual of your target operating system.

An error implies that the function where the error occurred exits. Clean-up is performed, which means that the application can continue its execution.

A special case to consider is when an error occurs in the server thread function, which runs statically during the execution. The thread exits and must be restarted manually.

If `THREADED_TRACE` is defined, the execution of the TCP/IP adapter is logged onto stdout when signals are encoded, sent, received and decoded.

Example 558: TCP/IP Adapter Error Message ––––––––––––––––––

An error occurs when a signal should be sent via the TCP/IP adapter. The following is logged on stdout:

```
ERROR xSendSignal/SCM_CONNECT: 146
```

The target platform is Solaris. The error code indicates that the connection was refused, probably because a server can not be found at the specified address. `xSendSignal` exits and the application continues its execution.

Configuration

The TCP/IP adapter is configured using compilation flags. The basic configuration can be done using the TCP/IP Connection Wizard in the Targeting Expert.

Including the TCP/IP adapter

The TCP/IP adapter requires environment files, environment header files and ASCII encoding/decoding for correct operation. These options are activated when the TCP/IP Signal Sending check box is enabled in the Targeting Expert's TCP/IP Connection Wizard.

Server Port Number

For a component that receives signals, a TCP port number must be specified. The server thread uses this port number to listen for incoming connections.

The port number is set in a text box in the TCP/IP Connection Wizard. You can also set the port manually by setting the flag `XSERVPORT` to the desired value. If no port number is specified, the port number is set to 5000 by default.

Routing

You must manually implement a routing function, so that a destination is specified for every SDL signal that is sent to the environment. The function must be declared in a C header file. The file is specified in a text box in the TCP/IP Connection Wizard or by setting the flag `XROUTING_INC` to the header file name.

The routing function implementation should be placed in a C file that is compiled and linked with the other code. It is specified using either the text box in the TCP/IP Connection Wizard or by including it in the compiler settings manually. See Example 557 for an example of how a routing function is implemented.

Using the TCP/IP Adapter with Other Environment Functionality

The TCP/IP adapter header file (`tcpipcomm.h`) is included from the environment file of a component. The file `tcpipcomm.h` contains `#define` statements for macros in the environment functions that invoke TCP/IP functions.

If you want to use other functionality in parallel with the TCP/IP adapter, these macros can be defined outside `tcpipcomm.h`. By setting the flag `XEXTENV_INC` to the header file you wish to use, your header file is included from `tcpipcomm.h`. This enables you to insert your code without modifying `tcpipcomm.h`. Please note that the `XEXTENV` flag can not be set in the TCP/IP Connection Wizard. It must be set manually.

Before using the `XEXTENV_INC` flag, look carefully at the #define statements in `tcpipcomm.h`. These must be valid for proper operation of the TCP/IP adapter.

Hint: Using External env Code with the TCP/IP Adapter

When combining the TCP/IP adapter with other environment functions, always preprocess the environment file to verify that the code is expanded as expected.

Example 559: External Code in Combination with the TCP/IP Adapter

A file called `mycomm.h` contains declarations of functions that should be used in parallel with the TCP/IP adapter. From the xInitEnv function, an initialization function should be called (`InitComm()`). From the `xInEnv` function, a function for polling communication (`ReadComm()`) should be invoked.

In `mycomm.h`, the following is inserted:

```
#define XENV_INIT Initcomm();\
                  xInitSignalSender();\
                  xInitSignalReceiver();
#define XENV_IN_START ReadComm();
```

`xInitsignalSender` and `xInitSignalReceiver` are taken from `tcpipcomm.h`, which contains the following:

```
#ifndef XENV_INIT
#define XENV_INIT xInitSignalSender();\
                  xInitSignalReceiver();
#endif
#ifndef XENV_IN_START
#define XENV_IN_START
#endif
```

The #defines in `tcpipcomm.h` are overridden. Still, the original calls are invoked, which ensures that the TCP/IP adapter executes properly.

Using Thread Parameters

Some thread parameters in the TCP/IP adapter can be set to fine-tune performance of your Threaded SDL application. The TCP/IP adapter threads do not use OS queues. Two parameters can be set: Thread Priority and Thread Stack Size. These are set manually using flags. They can not be set using the TCP/IP Communication Wizard.

The server wait thread uses the following flags:

```
XSERVTHRPRIO
XSERVTHRSTACK
```

The signal receiver threads use the following flags:

```
XRECVTHRPRIO
XRECVTHRSTACK
```

Set the flags to values that are specific to the target platform used.

New threaded integrations

Overview

The threaded integrations with Real-time operating systems described in this document has been developed by IBM Rational in the IBM Rational Tau product. Our intention, however, is that the integrations should be possible to use both in SDL Suite (Cadvanced) and Tau/Developer (AgileC and C Code Generator) products.

The difference between the currently existing threaded integrations and new ones presented here is not that large. The integration principles are almost exactly the same. The major difference is that the current integrations are expressed using macros, while the new ones uses a functional interface. Another difference is that all the current integrations provided by IBM Rational are mixed into one .h file, while in the new integration, each RTOS integration is implemented using one .h and one .c file. Both these changes are made to increase the readability and to simplify debugging. The change in file structure makes it also easier to implement new integrations and for a customer to modify an integration.

The currently available integrations are listed below.

Current integrations:

- SUN Solaris (This is a POSIX pthread integration that probably can be used on most UNIX-like operation system.)
- Win32
- VxWorks
- OSE

The new integrations:

- POSIX pthreads (tested on SUN Solaris and Linux but can probably be used on most UNIX-like operation system.)
- Win32
- VxWorks
- Nucleus Plus

Each RTOS integration consists of two files with the names rtapidef.h and rtapidef.c. The rtapidef.h file is included in the scttypes.h file, while the rtapidef.c file is included in the sctsdl.c file. As rtapidef.c is included in sctsdl.c there are no new files to compile and the make files are not effected, except for some compilation options discussed below.

When it comes to compilation none of the compilation switches used for the current threaded integrations should be defined. To use the new integrations the following should be given:

- a switch selecting an application kernel, for example SCTAPPLCLENV
- the switch THREADED
- a compiler option to tell the compiler where to find the rtapidef.h and the rtapidef.c file.

Example 560 ----------

```
cc -DSCTAPPLCLENV -DTHREADED -I/some/suitable/path
```

In the remaining part of this document the new threaded integrations are described in detail. This documentation is provided for customer who want to understand and possibly modify the integrations or to make new integrations themselves. The documentation was initially written for the

AgileC code generator in Tau/Developer and has been somewhat adopted to Cadvanced in SDL Suite.

Threaded integrations

To implement a new integration or to understand an existing one it is recommended to use this manual together with the code for some existing integration(s). There are some major aspects that have to be handled to implement an integration with real-time operating system.

- It is necessary to implement a clock function.
- There is need for a number of mutexes or binary semaphores to protect some shared data.
- Some startup code, for creating threads with relevant properties and synchronizing them are needed.
- A thread must be able to suspend its execution when it has nothing to do. It must then be possible to wake it up again when a signal is sent to a part in the thread.

To explain the details in these integration aspects the POSIX integration will be used as an example. Apart from the code mentioned below the rtapidef.h should include the necessary system include files to be able to access the concepts needed.

Example 561: Includes in rtapidef.h for POSIX ––––––––––––––––––

```
#include <pthread.h>
#include <sched.h>
#include <semaphore.h>
#include <time.h>
#include <sys/time.h>
```

––

If the RTOS has any requirements on the main function, which might be the case, it is possible to rename the main function included in uml_kern.c by defining XMAIN_NAME to for example:

```
#define XMAIN_NAME agilec_main
```

Then the user has to implement a proper main function that calls the `agilec_main` function.

The clock function

To support the SDL concept of timers, a clock function is necessary. The generated code and the kernel assumes that there is a clock function called `xNow` that returns the current time. Time values are represented by values of type `SDL_Time` which is defined as:

```
typedef struct
  s, ns xint32;
} SDL_Time;
```

`xint32` is implemented as a 32-bit int. The components `s` and `ns` represent the number of seconds and nanoseconds passed from some time in the past depending on the implementation of the clock function.

There are two standard implementations of the clock function, one for UNIX like systems and one for Windows. In Windows the standard function `_ftime` is used to read the system clock, while on UNIX like systems the standard function `clock_gettime` is used.

To implement a clock function you should include code in your own `rtapidef.h` and `rtapidef.c` files according to the details below.

If timers are not used and the clock is not explicitly accessed in SDL or C, there is no need for a clock implementation. Just include the macro definition:

```
#define xInitSystime()
```

in `rtapidef.h`.

If a clock implementation is needed then include the following prototypes in `rtapidef.h`:

```
extern void xInitSystime(void);
extern SDL_Time xNow (void);
```

If no initialization function is needed then the `xInitSystime` function can be replaced by the macro.

```
#define xInitSystime()
```

In the file `rtapidef.c` the implementation of these functions should be provided. The implementations will depend a lot on the support in software and hardware for the underlying architecture.

Protection of shared data

It is necessary to protect the list of available signals, the list of available timers, and a few other things. For this four global mutexes or binary semaphores are needed. These variables should be defined `extern` in `rtapidef.h` and declared in `rtapidef.c`. The names of the variables should be the same as in the example given below.

Example 562: In rtapidef.h:

```
extern pthread_mutex_t xFreeSignalMutex;
extern pthread_mutex_t xFreeTimerMutex;
extern pthread_mutex_t xCreateMutex;
#ifdef USER_CFG_USE_MEMORY_PACK
  extern pthread_mutex_t xMemoryMutex;
#endif
```

Example 563: In rtapidef.c:

```
pthread_mutex_t xFreeSignalMutex;
pthread_mutex_t xFreeTimerMutex;
pthread_mutex_t xCreateMutex;
#ifdef USER_CFG_USE_MEMORY_PACK
  pthread_mutex_t xMemoryMutex;
#endif
```

These four variables should be initialized during the startup of the application to an unlocked state. The function `xThreadInit` is a proper place for this initialization. Note that the names of the variables are chosen for AgileC in Tau/Developer, and do not fully reflect their usage in Cadvanced.

Example 564: xThreadInit

```
void xThreadInit (void)
{
  (void)pthread_mutex_init(&xFreeSignalMutex, 0);
  (void)pthread_mutex_init(&xFreeTimerMutex, 0);
  (void)pthread_mutex_init(&xCreateMutex, 0);
  #ifdef USER_CFG_USE_MEMORY_PACK
    (void)pthread_mutex_init(&xMemoryMutex, 0);
  #endif
  ....
}
```

The lock and unlock operation must also be implemented for mutexes or binary semaphores. The following two functions should be implemented.

Example 565 ----------

In **rtapidef.h:**

```
extern void xThreadLock (pthread_mutex_t *);
extern void xThreadUnlock (pthread_mutex_t *);
```

In rtapidef.c:

```
void xThreadLock (pthread_mutex_t *M)
{
  (void)pthread_mutex_lock(M);
}

void xThreadUnlock (pthread_mutex_t *M)
{
  (void)pthread_mutex_unlock(M);
}
```

Startup phase - creating the threads

After some basic initialization the kernel will in the main function start the specified threads. For each thread the functions xThreadInitOneThread and xThreadStartThread will be called, where xThreadInitOneThread should perform some thread specific initialization and xThreadStartThread should start the thread. Each thread should run the function xMainLoop declared in the kernel. This is performed by using a wrapper function, xThreadEntryFunc, which is defined in the integration and is the function really started in the thread.

After all the threads have been started the function xThreadGo is called in the function main. Some more information on these functions are given below.

It is important that the started threads do not execute any SDL transitions before all threads are created. Therefore the xThreadEntryFunc will as first action wait on a semaphore. The xThreadGo function will when all threads are created release this semaphore.

Many functions has a pointer to type xSystemData as parameter. This contains the local information for the thread. Among other things it con-

tains a field of type `xThreadVars`, which should be defined in the RTOS integration.

Example 566 ----------

`xThreadVars` type in `rtapidef.h`

```
typedef struct {
  pthread_mutex_t SignalQueueMutex;
  pthread_cond_t  SignalQueueCond;
  pthread_t       ThreadId;
} xThreadVars;
```

where the two first fields will be discussed in the next section, and the ThreadId will be used during the startup phase to store the identity of the threads.

The code for the behavior described in this section should look something like the following example.

Example 567 ----------

In `rtapidef.h`:

```
extern sem_t xInitSem;
#if !defined(USER_CFG_USE_xInEnv) && !defined(XENV)
  extern sem_t xMainThreadSem;
#endif

extern void xThreadInitOneThread (
   struct _xSystemData *);
extern void xThreadStartThread (
   struct _xSystemData *,
   unsigned int, unsigned int,
   unsigned int, unsigned int);
```

In `rtapidef.c`:

```
sem_t xInitSem;
#if !defined(USER_CFG_USE_xInEnv) && !defined(XENV)
  sem_t xMainThreadSem;
#endif

void xThreadInit (void)
{
  ....
  (void)sem_init(&xInitSem, 0, 0);
}

void xThreadInitOneThread(struct _xSystemData *xSysDP)
```

```
{
  (void)pthread_mutex_init(
    &xSysDP->ThreadVars.SignalQueueMutex, 0);
  (void)pthread_cond_init(
    &xSysDP->ThreadVars.SignalQueueCond, 0);
}


static void *xThreadEntryFunc (void *xSysDP)
{
  (void)sem_wait(&xInitSem);
  (void)sem_post(&xInitSem);
  xMainLoop((xSystemData *)xSysDP);
}


void xThreadStartThread(struct _xSystemData *xSysDP,
                        unsigned int StackSize,
                        unsigned int Prio,
                        unsigned int User1,
                        unsigned int User2)
{
  pthread_attr_t Attributes;
  ....
  (void)pthread_create(&xSysDP->ThreadVars.ThreadId,
                       &Attributes, xThreadEntryFunc,
                       (void *)xSysDP);
  ....
}

void xThreadGo(void)
{
  (void)sem_post(&xInitSem);

  #if defined(USER_CFG_USE_xInEnv)
    xInEnv();  /* AgileC */
  #elif defined(XENV)
    xInEnv(xNow());  /* Cadvanced */
  #else
    (void)sem_init(&xMainThreadSem, 0, 0);
    (void)sem_wait(&xMainThreadSem);
  #endif
}
```

The `xInitSem` semaphore is used for synchronization of the threads. It is initialized in the beginning of `xThreadInit` to 0, that is to a blocking state. After that the `xThreadStartThread` once for each thread that is to be started. The function `pthread_create` will call the function given as third parameter (`xThreadEntryFunc`) with the void * parameter given as fourth parameter (the `xSysD` pointer) as parameter. `pthread_create` will also store the identity of the thread in the variable passed as first parameter. The second parameter is the properties of the thread. This will be discussed later in this section.

If any of the threads get a chance to execute before all the threads are created, these threads will hang on the `sem_wait` call in `xThreadEntryFunc`, until the main thread calls `xThreadGo` that will post the semaphore `xInitSem` once. One of the threads waiting on this semaphore will then be able to execute and will immediately post the semaphore again. This will continue until all threads are free to execute.

After that all threads are running and depending on the OS and the application properties, the main thread can perform different things. Our recommendation is to call the `xInEnv` function and let that function run in this thread. For more details see the discussion on `xInEnv`. Another alternative is to hang the main thread on a semaphore, as shown above using the semaphore `xMainThreadSem` (if `xInEnv` is not used). In this case you can post the `xMainThreadSem` semaphore anywhere to restart the execution of the main thread.

When the main thread returns from the function `xThreadStart`, the program will continue to execute in the main function and will perform a call to exit. The behavior of a threaded program when the main thread performs exit, is OS dependent. In POSIX pthreads all threads are stopped at such an action. That is the reason it is important to hang the main thread at the end of the `xThreadStart` function.

Now to the properties of the threads. In most RTOS, properties like stack size and priority can be set for individual threads. Four integer values can be specified.

- The first value is interpreted as the stack size.
- The second value is interpreted as the priority.
- The third and fourth values can be used for other properties, defined by the RTOS integration or defined by you.

The currently predefined integrations only makes use of the first and second values. These values are passed as parameters to the `xTreadStartThread` function.

How the properties are set up in detail depend on the RTOS. Please see the available integrations, in the function `xThreadStartThread`, for examples.

In `rtapidef.h` proper default values for the four `xThreadData` fields should be set up. These default values are used if no value is specified in the thread definition.

Example 568 ----------

```
#define DEFAULT_STACKSIZE     1024
#define DEFAULT_PRIO             0
#define DEFAULT_USER1            0
#define DEFAULT_USER2            0
```

Suspending and waking up threads

When a thread finds out that it has nothing more to do, at least just for the moment, it should suspend itself to make the processor available for other threads. The thread should then wake up again either when a timer has expired and needs to be handled, or when some other thread (including `xInEnv`) sends a signal that should be treated by the suspended thread.

To implement these features one mutex or binary semaphore is used together with some sort of conditional variable. We need the possibility to perform a condition wait, with or without a timeout. We need also a way to signal to a thread to wake up again. These two entities are needed for each thread and is therefore included in the `xThreadVars` struct mentioned earlier:

```
typedef struct {
  pthread_mutex_t SignalQueueMutex;
  pthread_cond_t  SignalQueueCond;
  pthread_t       ThreadId;
} xThreadVars;
```

The purpose of the `SignalQueueMutex` is to protect the signal queue where signals from the outside of the thread are inserted. The `SignalQueueCond` should facilitate the conditional wait.

The `SignalQueueMutex` should be initialized in `xThreadInitOneThread`. If `SignalQueueCond` needs to be initialized it could be performed at the same place.

Example 569 ----------

```
void xThreadInitOneThread(struct _xSystemData *xSysDP)
{
 (void)pthread_mutex_init(&xSysDP-
>ThreadVars.SignalQueueMutex, 0);
 (void)pthread_cond_init(&xSysDP->ThreadVars.SignalQueueCond,
0);
}
```

The SignalQueueMutex is locked by using the function xThreadLock, discussed above. It is then unlocked in three different ways:

- xThreadUnlock (discussed above)
- xThreadWaitUnlock
- xThreadSignalUnlock

The xThreadWaitUnlock is called by the thread itself when it has come to the conclusion that it should suspend itself, while xThreadSignalUnlock is called by another thread that wants to wake up the current thread. Both functions are passed the xSysD pointer for the thread that the operation should be performed on.

Example: In rtapidef.h:

```
extern void xThreadWaitUnlock (struct _xSystemData *);
extern void xThreadSignalUnlock (struct _xSystemData
*);
```

Example: In rtapidef.c:

```
void xThreadWaitUnlock (struct _xSystemData *xSysDP)
{
  #if defined(CFG_USED_TIMER) || defined(THREADED)
  #ifdef THREADED
      /* Cadvanced */
    if (xSysDP->xTimerQueue->Suc==xSysDP->xTimerQueue)
{
  #else
      /* AgileC */
    if (! xSysDP->TimerQueue) {
  #endif
      (void)pthread_cond_wait(
        &xSysDP->ThreadVars.SignalQueueCond,
        &xSysDP->ThreadVars.SignalQueueMutex);
    } else {
      struct timespec timeout;
      #ifdef THREADED
          /* Cadvanced */
        timeout.tv_sec =
          ((xTimerNode)xSysDP->xTimerQueue->Suc)->
            TimerTime.s;
        timeout.tv_nsec =
          ((xTimerNode)xSysDP->xTimerQueue->Suc)->
            TimerTime.ns;
      #else
          /* AgileC */
        timeout.tv_sec = xSysDP->TimerQueue->Time.s;
        timeout.tv_nsec = xSysDP->TimerQueue->Time.ns;
      #endif
      (void)pthread_cond_timedwait(
```

```
            &xSysDP->ThreadVars.SignalQueueCond,
            &xSysDP->ThreadVars.SignalQueueMutex,
            &timeout);

      }
    #else
      (void)pthread_cond_wait(
        &xSysDP->ThreadVars.SignalQueueCond,
        &xSysDP->ThreadVars.SignalQueueMutex);
    #endif
    (void)pthread_mutex_unlock(
      &xSysDP->ThreadVars.SignalQueueMutex);
}

void xThreadSignalUnlock (struct _xSystemData *xSysDP)
{
  (void)pthread_cond_signal(
    &xSysDP->ThreadVars.SignalQueueCond);
  (void)pthread_mutex_unlock(
    &xSysDP->ThreadVars.SignalQueueMutex);
}
```

At the time when `xThreadWaitUnlock` or `xThreadSignalUnlock` is called the `SignalQueueMutex` will be locked and must therefore be unlocked at the end of both functions.

In `xThreadWaitUnlock` the thread wants to suspend itself. If timers are used and there is a timer active in the timer queue, it should wait until the timer expires or until some other thread tells it to wake up. In POSIX pthreads the function `pthread_cond_wait` performs exactly this. If timers are not used or there is no timer active, the thread should be suspended until someone else wakes it up. In POSIX pthreads this can be achieved with the function `pthread_cond_wait`.

In `xThreadSignalUnlock` the thread given by the parameter should be waken up. Here the `pthread` function `pthread_cond_signal` can be used.

The integrations described here are also used when the Cadvanced is used to generate threaded applications. This adds a few requirements in the implementation of a threaded integration. First a function that can stop a thread is needed.

In rtapidef.h:

```
#if defined(THREADED) || defined(CFG_USED_DYNAMIC_THREADS)
extern void xThreadStopThread(struct _xSystemData *);
#endif
```

In rtapidef.c:

```
#if defined(THREADED) || defined(CFG_USED_DYNAMIC_THREADS)
void xThreadStopThread(struct _xSystemData *xSysDP)
{
  pthread_mutex_destroy(&xSysDP->ThreadVars.SignalQueueMutex);
  pthread_cond_destroy(&xSysDP->ThreadVars.SignalQueueCond);
  pthread_exit(0);
}
#endif
```

The `xThreadStopThread` function should clean up the thread specific semaphores and stop the thread. It is always the thread that should be stopped that will call this function to stop itself.

Another difference is the way timers are accessed for the two code generators. This effects the details in the `xThreadWaitUnlock` function. Please see this function above and especially the sections under #ifdef THREADED.

In the case of the Cadvanced the RTOS integrations are accessed through a macro layer. The macros in this layer is used in the Cadvanced kernel files and in the generated code.

Example 570: Defines in scttypes.h ––––––––––––––––––––––––––

The following defines are relevant (from scttypes.h):

```
#define THREADED_GLOBAL_VARS
#define THREADED_GLOBAL_INIT \
     xThreadInit();
#define THREADED_THREAD_VARS \
     xThreadVars ThreadVars;
#define THREADED_THREAD_INIT(SYSD) \
     xThreadInitOneThread(SYSD);
#define THREADED_THREAD_BEGINNING(SYSD)
#define THREADED_AFTER_THREAD_START \
     xThreadGo();
#define THREADED_START_THREAD(F, SYSD, STACKSIZE,
PRIO, USER1, USER2) \
xThreadStartThread(SYSD, STACKSIZE, PRIO, USER1,
USER2);
#define THREADED_STOP_THREAD(SYSD) \
     xThreadStopThread(SYSD);
#define THREADED_LOCK_INPUTPORT(SYSD) \
     xThreadLock(&SYSD-
>ThreadVars.SignalQueueMutex);
#define THREADED_UNLOCK_INPUTPORT(SYSD) \
     xThreadUnlock(&SYSD-
>ThreadVars.SignalQueueMutex);
#define THREADED_WAIT_AND_UNLOCK_INPUTPORT(SYSD) \
     xThreadWaitUnlock(SYSD);
#define THREADED_SIGNAL_AND_UNLOCK_INPUTPORT(SYSD) \
     xThreadSignalUnlock(SYSD);
#define THREADED_LISTREAD_START
```

```
xThreadLock(&xFreeSignalMutex);
#define THREADED_LISTWRITE_START
xThreadLock(&xFreeSignalMutex);
#define THREADED_LISTACCESS_END
xThreadUnlock(&xFreeSignalMutex);
#define THREADED_EXPORT_START
xThreadLock(&xCreateMutex);
#define THREADED_EXPORT_END
xThreadUnlock(&xCreateMutex);
```

Tight Integration

Note:

The source file and examples for Tight Integrations are not included in the standard delivery. They are available as free downloads from IBM Rational Support web site.

Note:

This presentation is focused on the general principles and models used in a Tight Integration. When specific RTOS primitives are needed in the presentation, examples from the VxWorks implementation are used. The implementation and RTOS calls used in other integrations are covered in separate annexes to this chapter, one for each supported RTOS.

There are two models of Tight Integration. In the Standard Model one SDL process instance is mapped to one OS task. In the Instance Set Model an entire instance set (all instances of a process) is mapped to one OS task. Scheduling between OS tasks is managed by the RTOS scheduler; this means that preemption is normally used, though only on an instance set level in the Instance Set Model. SDL semantics are preserved in a Tight Integration, for example setting a timer implies an automatic reset first.

The start-up of a system, i.e. creation of static processes, initialization of synonyms and creation of an environment task and a timer task, is handled by a generated initialization function called `yInit`. Normally this function is called from another initialization function, where some additional initializations take place before the `yInit` function is called.

Timers in the system are handled by one central timer task. This task receives messages[1], each containing a request to set a timer, and will send messages back as the timers expire.

Common Features

File Structure

The files related to the tight integration concept are placed in the following directory in the installation: `<installation directo-`

1. SDL signals will be implemented as messages in VxWorks.

ry>/sdt/sdtdir/RTOS/<operating system>/TightIntegration/. The same files are used for both the Standard Model and the Instance Set Model.

Each RTOS directory contains the following files:

- sct<RTOS>.h:

  This file contains the second level of macros (see the comments for scttypes.h in "The Integration Packages" on page 3291). All macros are using OS-specific calls or types.

- sct<RTOS>.c:

  This file contains OS-specific support functions.

- sdt2<RTOS>.c:

  Most RTOS require that signals/messages are represented with an integer value. This is the source file for a utility program for generating signal identities. Each signal will be assigned an integer value. The output will be a file with the suffix .hs. This file is automatically included in the application.

  In the SDL Suite, the .hs file can also be generated by the SDL to C Compiler by turning on the option *Generate signal number file* in the *Make* dialog. The .hs file is included in the application if the compilation switch XINCLUDE_HS_FILE is set.

The SDL_PId Type

The SDL_PId (SDL Process ID) type has different meanings in the Standard and the Instance Set Models. In the Standard Model it represents the message queue, while it represents the process instance in the Instance Set Model. This is because the entire instance set will have the same message queue in the last case.

```
#ifdef X_ONE_TASK_PER_INSTANCE_SET
typedef xEPrsNode SDL_PId;
#else
typedef MSG_Q_ID  SDL_PId;
#endif
```

Signals

The signal header consists of a struct with information needed to handle the signal inside an SDL system. The signal header struct is defined in the RTOS-specific file `sct<RTOS>.h`.

```
typedef struct xSignalHeaderStruct *xSignalHeader;
typedef struct xSignalHeaderStruct {
  int            SignalCode;
  xSignalHeader  Pre, Suc;
  SDL_PId        Sender;
  void          *SigP;
  #ifdef X_ONE_TASK_PER_INSTANCE_SET
    SDL_PId        Receiver;
  #endif
  #ifdef XMSC_TRACE
    int            SignalId;
    int            IsTimer;
  #endif
} xSignalHeaderRec;
```

The signal header stores `SignalCode`, in this case an integer, two pointers `Pre` and `Suc` used when saving the signal in the save queue, and `Sender`, holding the `SDL_PId` of the sending SDL process. In the Instance Set Model there is an extra parameter `Receiver`, necessary to make a distinction between the SDL processes in an instance set task.

If the signal contains parameters they are allocated in the same function call. Example:

```
OutputSignalPtr = xAlloc (sizeof (xSignalHeaderRec)
+ sizeof(yPDef_z05_s2));
```

The second parameter to the `xAlloc` function is a struct representing the signal parameters of the signal. In this case, with one integer, it is defined in the following way:

```
typedef struct {
  SIGNAL_VARS
  SDL_Integer     Param1;
} yPDef_z05_s2;
```

The macro `SIGNAL_VARS` is in most RTOS empty.

There is an extra element in the `SignalHeader` defined as a `void` pointer. This pointer `SigP` is set to point to the parameter area.

```
OutputSignalPtr->SigP = OutputSignalPtr+1;
```

This pointer is used in the Signal-Free-Function to address the parameter-part of either a signal-structure as also a timer-signal-structure.

Note:

The SDL signal parameters are always named `Param1`, `Param2`, etc.

Assignment of the signal parameter is done in generated code and not in a macro. Example:

```
SIGNAL_ALLOC_ERROR
yAssF_SDL_Integer(((yPDef_z05_s2*)
  OUTSIGNAL_DATA_PTR)->Param1 ,yVarP->z023_param1,
  XASS);
```

The macro `OUTSIGNAL_DATA_PTR` macro is defined:

```
#define OUTSIGNAL_DATA_PTR (yOutputSignalPtr->SigP)
```

After expansion of the whole expression the code will be:

```
((yPDef_z05_s2 *) ((xSignalHeader) yOutputSignalPtr
+ 1))->Param1 = yVarP->z023_param1;
```

Signal reception

The support function `xInputSignal` is used for receiving signals in both models of Tight Integration. The implementation and the parameters are different though.

Timer Signals

A timer signal is defined similarly to an ordinary signal but will contain some additional elements representing time-out time, etc. The timer header struct looks like this:

```
typedef struct xTimerHeaderStruct *xTimerHeader;
typedef struct xTimerHeaderStruct {
      int             SignalCode;
      xTimerHeader    Pre, Suc;
      SDL_PId         Sender;
      void           *SigP;
#ifdef X_ONE_TASK_PER_INSTANCE_SET
      SDL_PId         Receiver;
#endif
#ifdef XMSC_TRACE
      int             SignalId;
      int             IsTimer;
#endif
      SDL_Time        TimerTime;
```

```
        int             TimerToSetOrReset;
        xbool           (* yEq)();
        xbool           TestParams;
        xTimerHeader    Param;
} xTimerHeaderRec;
```

Note:

An ordinary signal is identical to the first part of a timer signal. This makes it possible to type-cast between the two types as long as only elements in the common part of the headers are used.

Time

When the System time is required, for example when using NOW, the macro `SDL_NOW` is used. The macro is in turn mapped to the function `SDL_Clock()` (in `sctos.c`). This function is implemented differently depending on the RTOS representation of time. In VxWorks it returns the result of calling the RTOS function `tickGet`. `SDL_Time` is normally implemented as `int` or `unsigned long int`.

Mapping Between SDL Time and RTOS Time

The macro `SDL_DURATION_LIT` specifies the mapping between the SDL time in seconds and the local RTOS representation of time. In VxWorks the system time is given in ticks and the translation is defined as follows:

```
#define SDL_DURATION_LIT(R,I,D) \
  ((I)*1000 + (D)/1000000)
```

`R` is the real type representation of the time in seconds. `I` and `D` are the integer and decimal parts of an integer type representation of the time. `I` is in seconds and `D` in nanoseconds. The code generator will generate all three numbers but either `R`, or `I` and `D` will be used depending on the RTOS.

Timers

All timer activity in the SDL system is handled by a dedicated timer task. The timer task accepts requests in the form of messages (in VxWorks). It then keeps the requests for setting a timer sorted in a timer queue and uses some OS mechanism to wait for the first request to time out. The mechanism used can be either an OS timer, or a timeout in the waiting for new requests. When a request times out, the timer task sends a signal back to the task that first sent the request. The function calls and

OS signaling involved in setting and waiting for an SDL timer can be viewed in Figure 573.

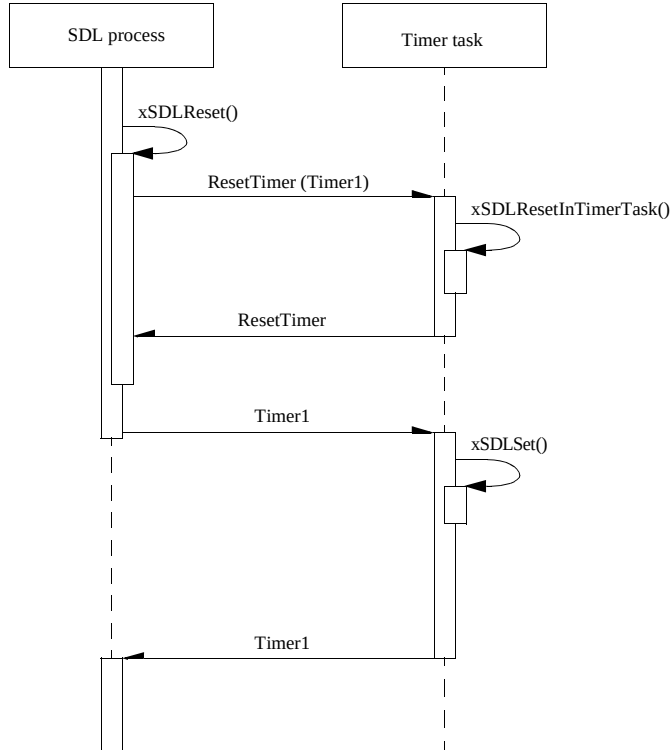


Figure 573: Function calls and OS signaling when setting and waiting for a timer.

UML notation is being used, thus the full arrowheads represent function calls and the half arrowheads represent messages. Parameters have been left out, except for the name of the timer in the ResetTimer request. Note that a reset is performed first, as required in the SDL specification.

To be able to implement the full semantics of SDL timers a number of support functions have been implemented:

- `xSDLActive`

  Checks whether an SDL timer is active and returns true or false. The function passes the question on to the timer task in the form of a request.

- `xSDLActiveInTimerTask`

  Called by the timer task upon request. Checks if an SDL timer is active and returns true or false.

- `xSDLReset`

  Resets an SDL timer by sending a request to the timer task. While waiting for a reply all new signals to the calling task are saved in the save queue. In the case of an Instance Set Model Tight Integration this means that no instance of the process can execute a transition until a reply is received. If the reply states that the timer couldn't be found it might be in the save queue or the input queue of the task because it has recently expired. If so, it is simply removed. SDL semantics require that a reset is always performed implicitly prior to setting a timer.

- `xSDLResetInTimerTask`

  Called by the timer task when a request has been made for resetting a timer. Checks the timer queue to see if the timer to reset is there. If the timer is found, it is removed and the data area it holds is freed. A message is sent back to the task that made the request, telling whether the timer was found or not.

- `xSDLSet`

  Called by the timer task when a request has been made for setting a timer. This function sorts the request into the timer queue.

Addressing SDL Processes

There are two ways to address SDL processes from an external task. Either the `xFindReceiver` function can be called to find an arbitrary receiver, or the file `pidlist.pr` can be used to provide a list of the SDL processes and then address the receiver explicitly via the input queue ID of its OS task.

The xFindReceiver Function

When sending a signal into the SDL system where the receiver is not known a support function called `xFindReceiver` can be used. This function takes the following parameters:

- The ID of the signal

- The sender ID (in this case an SDL_PId representing the environment)
- An optional VIA-list.

The following files are needed to get access to SDL types, signal numbers and signal parameter types: `scttypes.h`, `<system_name>.hs` and `<system_name>.ifc`.

Example of how to use the `xFindReceiver` function:

```
#include "scttypes.h"
#include "<system_name>.ifc"
#include "<system_name>.hs"

void MyExtTask(void) {

   xSignalHeader yOutputSignalPtr;
   int Err;

   /*Allocate signal header and signal parameter
     buffer */
   yOutputSignalPtr =
      (xSignalHeader)xAlloc(sizeof(xSignalHeaderRec)
       + sizeof(yPDef_go);

   /*Setup signal header */
   yOutputSignalPtr->SignalCode = go;
   yOutputSignalPtr->Sender = xEnvPId;

   /*Give value 100 to integer parameter */
   ((yPDef_go *)(yOutputSignalPtr+1))->Param1 = 100;

   /*Send signal from environment */
   Err = msgQSend(xFindReceiver(go, xEnvPrs, 0),
           (char*) yOutputSignalPtr,
           sizeof(xSignalHederRec)+sizeof(yPDef_go),
           0 ,0);
}
```

The following types, signal definitions and global variables are used in the example:

- `xEnvPId`: An SDL PId representing the environment, from `scttypes.h`
- `xEnvPrs`: A PrsNode representing the environment, from `scttypes.h`
- `xSignalHeader`: A datatype representing an SDL signal, from `scttypes.h`

- `yPDef_go`: A datatype representing the signal parameter types, from `<system_name>.ifc`
- `go`: An SDL signal, from `<system_name>.ifc`

The File pidlist.pr

An alternative way to get the PId for the Receiver is to use an ADT defined in the ADT library called `pidlist.pr`. This file defines an ADT called PidList and an operator called PId_Lit. With this ADT it is possible to directly address any static process instance in the system, both from internal SDL processes and from external OS-tasks. You can find more information about this feature in "How to Obtain PId Literals" on page 3256 in chapter 62, *The ADT Library*.

> **Note:**
>
> If you need the `pidlist.pr` ADT in a Tight Integration then you must use the version in the `<installation directory>/sdt/sdtdir/RTOS/SDL/` directory.

The Standard Model

In the Standard Model of the Tight Integration each SDL process is implemented as an OS task. Preemption and the use of process priorities is only limited to what the OS supports.

Processes

Process Creation

An SDL process is created in the following way (in the VxWorks integration):

1. A start-up signal is allocated.
2. A message queue is created. Some operating systems create the message queue automatically when the task is created. This is explained for each operating system in the annexes to this chapter.
3. The task is created with the message queue ID as a start-up parameter. In the case of VxWorks, the task will have a name starting with `VXWORKSPAD_`. This is a function which will first initialize some internal variables and then call the PAD function.

4. A function (`xAllocPrs`) is called to create a representation of the new instance in the global symbol tree.
5. The start-up signal is sent. When this signal is received in the task the start transition of the process is executed.

Process Termination

The following actions are carried out when a process terminates:

1. The save queue and the message queue are emptied.
2. The save queue is deleted.
3. A message is sent to the `xTimerTask` with a request to remove all active timers of the process.
4. `xFreePrs` is called to free the `PrsNode`.
5. The message queue is deleted. In some operating systems this is done automatically when the task is deleted.
6. The task is deleted.

PAD functions

Each PAD (Process Activity Definition) function will contain an eternal loop with an OS receive statement. When a process instance is created it is the PAD function that is called in the OS Create primitive.

The start-up and execution of a PAD function works like this:

1. The support function `xInputSignal` is called. This function will wait for the start-up signal, that is always received first, and then return to the PAD function.
2. The PAD function goes to the label `Label_Execute_Transition`. This label is the start of a code block containing a switch statement that evaluates the process variable `RestartAddress`. The code under each different case then represents a transition. At the end of this block the process variable `State` is updated and execution continues at `Label_New_Transition`.
3. In `Label_New_Transition` a new call is made to `xInputSignal` and execution then continues at `Label_Execute_Transition`.

The structure of a PAD function is described below (with pseudo-code shown in *italics*):

```
void yPAD_z01_pr1 (void *VarP)
{
  Variable declarations
  xInputSignal is called to receive the start-up signal
  ...........
  goto Label_Execute_Transition;
  ...........
  Label_New_Transition:
  xInputSignal is called to receive a signal

  Label_Execute_Transition:
  Local declarations
  switch (yVarP->RestartAddress) {
  case 0:
    Execute the start transition
    Update the process state variable
    goto Label_New_Transition;
  case 1:
    Execute the transition
    Update the process state variable
    goto Label_New_Transition;
  ...........
  }
  ...........
}
```

Scheduling

Since each SDL process is implemented as an OS task, scheduling between processes will be handled completely by the OS.

Start-up

Start-up of a Standard Model Tight Integration can be described as follows (pseudocode is shown in *italics*):

```
MyMain() {
/* initialization of semaphores etc */
 yInit();
 Give startup semaphores
 taskSuspend(Mymain);
}

yInit() {
 Create the timer task
 Create an environment task or only an environment queue
 for(i=1;i<=NoOfStaticProcessTypes;i++){
  for(j=1;j<=NoOfStaticInstancesOfEachProcesstype;
      j++){
    Allocate a startup signal
    Create a message queue
```

```
        Create a task
        Call xAllocPrs
        Send the startup signal
      }
    }
    Assign SDL synonyms
  }
```

The semaphore is used for synchronizing start-up of static processes. No static process is allowed to execute its start transition before all static processes are created, because a start transition can have signal sending to other static instances.

The `MyMain` function is placed among other support functions in the file `sctsdl.c`.

The `yInit` function is generated by the code generator and placed last in the generated file for the system.

The Instance Set Model

The Instance Set Model is based on the same principles as the Standard Model with the difference that the instance set is the basic unit rather than the process instance.

Processes

Both the instances and the instance sets are represented in the symbol table. In addition to the three parts that always make up an SDL process there is also an extra struct for the instance set, defining for example the input queue which is common to all the instances. Further, there is a PAD function for each instance, but also for the instance set.

Process Representation

An SDL_PId is represented by an `xEPrsNode`, pointing to an `xEPrsStruct`. An `xEPrsNode` also represents a process instance in the symbol table both in the Standard Model and the Instance Set Model.

```
typedef struct xEPrsStruct {
  xEPrsNode       NextPrs;
  SDL_PId         Self;
  xPrsIdNode      NameNode;
  int             BlockInstNumber;
  xPrsNode        VarP;
} xEPrsRec;
```

Instance Set Data

The datatype `xPrsInstanceSetVars` is only used in the Instance Set Model. It defines common data for all instances of the set, like the save queue and the size of the instance data.

```
typedef struct {
  xSignalHeader   SaveQ;
  xSignalHeader   CurrentInSaveQ;
  xSignalHeader   yInSignalPtr;
  char            name[100];
  unsigned        PrsDataSize;
} xPrsInstanceSetVars.
```

PAD functions

In the Instance Set Model there is a PAD function for each process instance but also for each instance set. The instance set PAD functions will be called at system start-up and contain an eternal loop in the same fashion as PAD functions in the Standard Model. Instance PAD functions are only called to execute transitions.

Process Creation

All instance sets, even for dynamic processes, are created at system start-up. Since the OS task and the signal queues are created with the instance set, the creation of an instance requires less labor than in the Standard Model. For the instance set creation the macro `INIT_PROCESS_TYPE` is used.

Process Termination

The instance set task is never terminated. Termination of a process instance will not remove the save queue, the input queue and the task. This is done at system termination. All queues, including the active timer queue, are emptied of messages to the terminated process though.

Signal queues

The message queue id of the receiver's instance set is accessed through `NameNode` in the receiver's `xEPrsStruct` and the variable `PROCID`.

Example from `xSDLResetInTimerProcess`:

```
Err=msgQSend ((MSG_Q_ID) (yInSignalPointer->Sender)
  ->NameNode->PROCID, (char *) yInSignalPointer,
  sizeof (xTimerHeaderRec), 0, 0);
```

In this case the receiver is the same as the original sender.

Signal sending

A support function `xHandle_Sig` is used when sending signals, instead of the macro `RTOSSEND` as in the Standard Model. This difference is shown in **bold** in the code below:

```
#ifdef X_ONE_TASK_PER_INSTANCE_SET
#define SDL_2OUTPUT(PRIO, VIA, SIG_NAME, SIG_IDNODE,\
   RECEIVER, SIG_PAR_SIZE, SIG_NAME_STRING)\
   XOS_TRACE_OUTPUT(SIG_NAME_STRING) \
   XMSC_TRACE_OUTPUT(RECEIVER, yOutputSignalPtr, \
     SIG_NAME_STRING) \
   xHandle_Sig(yOutputSignalPtr,SDL_SELF,SIG_PAR_SIZE,\
    RECEIVER,(RTOSTASK_TYPE)RECEIVER-> \
    NameNode->PROCID RTOSHANDLESIG_PAR);
#else
#define SDL_2OUTPUT(PRIO, VIA, SIG_NAME, SIG_IDNODE, \
   RECEIVER, SIG_PAR_SIZE, SIG_NAME_STRING)\
   XOS_TRACE_OUTPUT(SIG_NAME_STRING) \
   XMSC_TRACE_OUTPUT(RECEIVER, yOutputSignalPtr,\
     SIG_NAME_STRING) \
   RTOSSEND(&yOutputSignalPtr, RECEIVER,SIG_PAR_SIZE)
#endif
```

Scheduling

Scheduling between instance sets is handled by the operating system. Within the instance sets, however, scheduling is based on the signal queue. When the instance set PAD function is executing, it takes the first signal in the input queue and calls the PAD function of the addressed SDL process. The instance PAD function then executes one transition and returns control to the scheduling loop of the instance set PAD function.

Integrating with external code

You can easily integrate the SDL system with external code, for example written in C. Just use the hooks described below for inserting C statements in the `main()` function of the SDL system.

The hooks are in the form of `#define` macros located in a file called `scthooks.h`. A file called `scthooks.h_template` with empty macros can be found in the `INCLUDE` directory. Use this file as a template for your own application. You will find usage examples in the `Examples` directory.

`HOOK_GLOBAL_DECLARATIONS`

This hook lets you declare function prototypes etc. at file scope.

HOOK_MAIN_DECLARATIONS

This hook lets you declare variables for use in the `main()` function.

HOOK_MAIN_START_OF_CODE

Any code inserted here will execute first in the `main()` function.

HOOK_MAIN_AFTER_PROCESS_RELEASE

Any code inserted here will execute as soon as all static processes have been created and are allowed to run.

HOOK_MAIN_AFTER_SIGNAL_RECEPTION

The `main()` function of the SDL system enters an infinite loop after having created all static processes. This loop is used to receive signals sent to the environment queue.

Use the `HOOK_MAIN_AFTER_SIGNAL_RECEPTION` to insert code for processing these signals.

Limitations for Integrations

In general, the same restrictions as for the SDL to C Compiler apply, but Tight integrations have some further restrictions. The detailed limitations for Light and Tight integrations are listed in the Release Guide.

A Simple Example

This section describes an example system named Simple. The annexes show how to integrate the example with different operating systems.

The example demonstrates the following techniques:

- How to integrate an SDL system with an operating system
- How to make the environment communicate with the SDL system in a Light integration
- How to make an external process written in C communicate with the SDL system in a Tight integration
- How to use the special Tight Integration version of the ADT `pidlist.pr` .

The Simple System

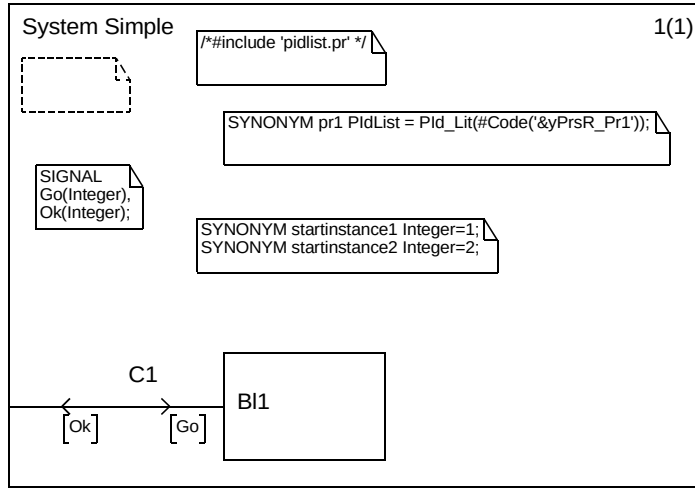


Figure 574: System Simple

The SDL representation of Simple consists of a single block Bl1. Seen from the outside, the system accepts the signal Go and responds after about five seconds by sending the signal Ok. The signal Go may be sent twice to the system.

Block Bl1

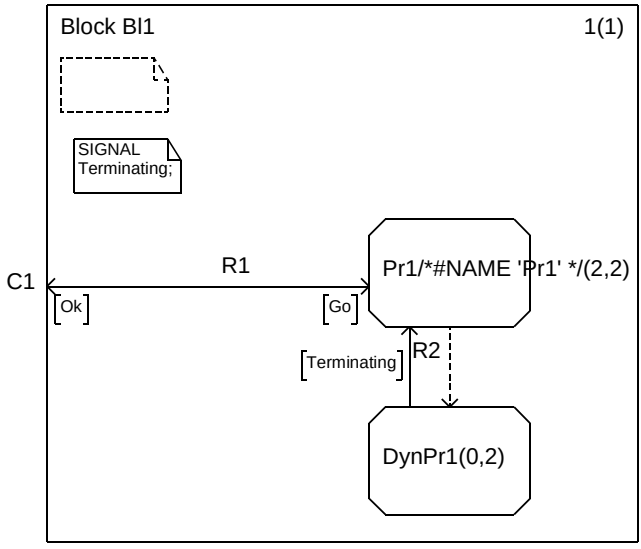


Figure 575: Block Bl1

Block Bl1 has two processes. The static process Pr1 and the dynamic process DynPr1. DynPr1 is created by Pr1 and can send the signal Terminating back to its parent. Pr1 handles all the interaction with the environment, through the signals Go and Ok.

Process Pr1

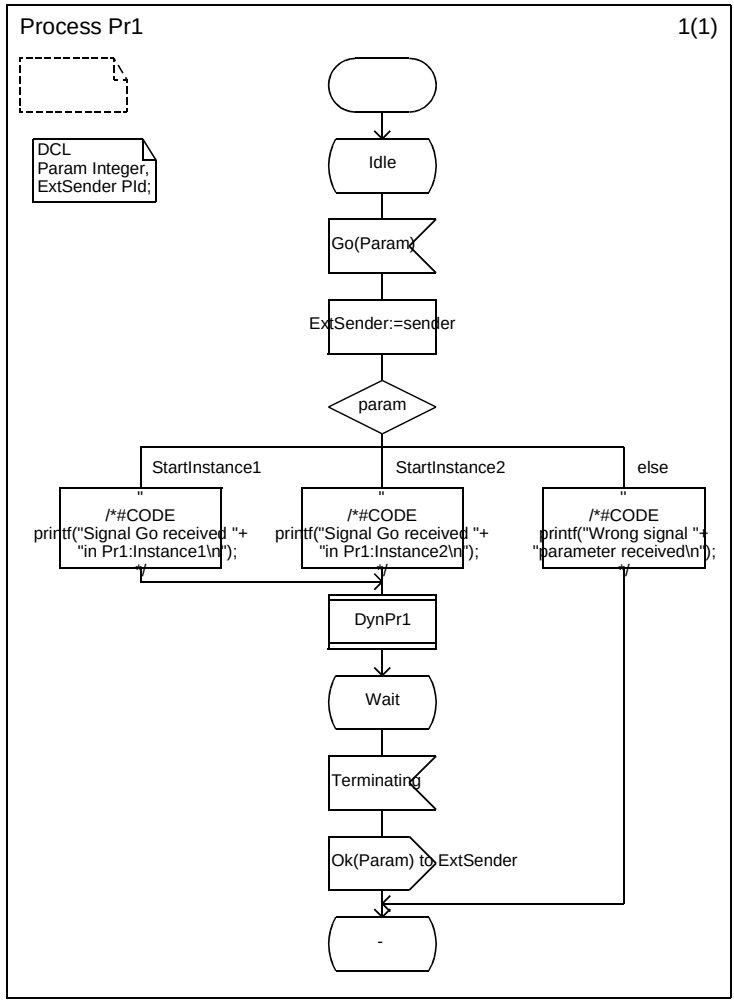


Figure 576: Process Pr1

Process Pr1 is a static process with two instances. It has two states, Idle and Wait. In the Idle state the process waits for the signal Go with an integer parameter representing the instance number of this instance. It then prints the instance number to the standard output, creates one in-

stance of DynPr1 and enters the Wait state. In the Wait state it waits for the signal Terminating from the created instance of DynPr1, sends Ok back to the environment and goes back into the Wait state. Since the Terminating signal will only be received once, the process is going to remain in the Wait state forever.

Process DynPr1

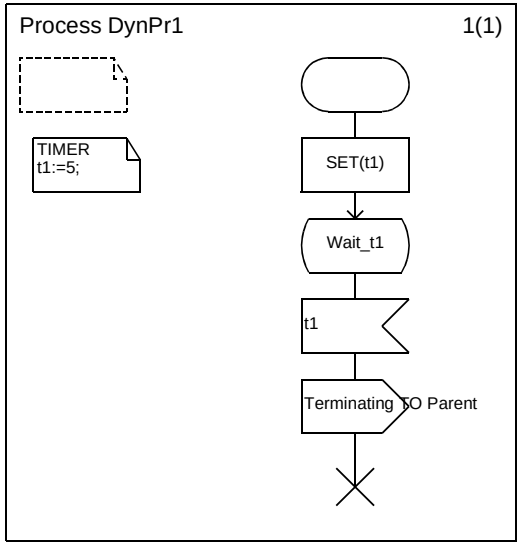


Figure 577: Process DynPr1

The dynamic process DynPr1 has no instances at system start and a maximum of two instances. Each instance of Pr1 creates one instance of DynPr1. DynPr1 sets the timer t1 to five seconds, waits for a timeout, sends the signal Terminating to its creator and finally terminates.

Connection to the Environment

The environment is handled in different ways depending on the integration model. See below for details.

Building and Running a Light Integration

This section will take you through the general steps required to build a Light Integration for the Simple example. The procedure works the same for most operating systems. Please also check the annexes for important information about your operating system.

General Steps for a Light Integration

1. Create a working directory and an INCLUDE directory below it.

> **Note:**
> `$(sdtdir) = <installation directory>/sdt/sdtdir/<machine dependent dir>`
>
> where `<machine dependent dir>` is `sunos5sdtdir` on SunOS 5 and `wini386` in Windows.

2. Copy all files from `$(sdtdir)/RTOS/Examples/Simple/<selected RTOS>/LightIntegration` to the working directory.
3. Copy the following files to the `INCLUDE` directory:

   ```
   $(sdtdir)/INCLUDE/sctlocal.h
   $(sdtdir)/INCLUDE/sctpred.c
   $(sdtdir)/INCLUDE/sctsdl.c
   $(sdtdir)/INCLUDE/sctos.c
   $(sdtdir)/INCLUDE/sctpred.h
   $(sdtdir)/INCLUDE/scttypes.h
   ```

4. Open the system file
   `$(sdtdir)/RTOS/Examples/Simple/simple.sdt`
5. Change the destination directory to your working directory
6. Set the options *Lower Case* and *Generate environment header file.*
7. Select the Cadvanced SDL to C Compiler and generate an application.
8. Edit the `makefile` supplied with the example to fit your environment. Normally you will only have to point out the directory where the RTOS is installed.
9. Use the makefile to build an executable.
10. Download the executable to target or run it under a kernel simulator (“soft kernel”).

Result From Running the System

The output when running the example should be:

```
Signal Go received in Pr1:Instance1
Signal Go received in Pr1:Instance2
Signal Ok received with the following parameter:1
Signal Ok received with the following parameter:2
```

The xInEnv Function

This is where the start-up signal Go is sent. In a real system `xInEnv` may be used for polling hardware devices for data. The code looks like this:

```
xSignalNode S;
static int SendGo = 0;

if(SendGo<=1){
  if(SendGo==0) {
    S = xGetSignal(go,  pr1[1],   xEnv );
    ((yPDef_go *)(S))->Param1 = startinstance1;
  }
  else {
    S = xGetSignal(go,  pr1[2],   xEnv );
    ((yPDef_go *)(S))->Param1 = startinstance2;
  }
  SDL_Output( S, xSigPrioPar(xDefaultPrioSignal)
                 (xIdNode *)0 );
  SendGo++;
```

The signal Go is sent the first and the second time `xInEnv` is called. The parameters `startinstance1 and startinstance2` are integer constants defined in the SDL system, as integer `SYNONYM`'s. They are made available by generating and including the file `simple.ifc`.

The xOutEnv Function

The code in `xOutEnv` for receiving the signal Ok looks like this:

```
if ( ((*S)->NameNode) == ok ) {
  printf("Signal Ok received with the following
          parameter:%lu\n",
          ((yPDef_ok *)(*S))->Param1);
  xReleaseSignal( S );
  return;
```

The signal Ok also has an integer parameter, the value of this should be 1 if it is sent by Pr1 instance one and 2 if it is sent by instance two.

Building and Running a Tight Integration

This section will take you through the general steps required to build a Tight Integration for the Simple example. The procedure works the same for most operating systems. Please also check the annexes for important information about your operating system.

Note:

The source file and examples for Tight Integrations are not included in the standard delivery. They are available as free downloads from IBM Rational Support web site.

General Steps for a Tight Integration

1. Create a working directory and an INCLUDE directory below it.

Note:

`$(sdtdir) = <installation directory>/sdt/sdtdir/<machine dependent dir>`

where `<machine dependent dir>` is `sunos5sdtdir` on SunOS 5 and `wini386` in Windows.

2. Copy all files from `$(sdtdir)/RTOS/Examples/Simple/<selected RTOS>/TightIntegration` to the working directory.
3. Copy all files from `$(sdtdir)/RTOS/Examples/Simple/<selected RTOS>/TightIntegration`/INCLUDE to the INCLUDE directory.
4. Open the system file `$(sdtdir)/RTOS/Examples/Simple/simle.sdt`
5. Change the destination directory to your working directory.
6. Set the options *Lower Case*, *Generate environment header file* and *Generate signal number file*.
7. Select the Cadvanced SDL to C Compiler and generate an application.
8. Edit the `makefile` supplied with the example to fit your environment. Normally you will only have to point out the directory where the RTOS is installed.

9. Use the makefile to build an executable.
10. Download the executable to target or run it under a kernel simulator ("soft kernel").

Result From Running the System

The output when running the example depends on what kind of trace has been enabled. If you set XMSC_TRACE it should be similar to the following:

```
System_Init_Proc: instancehead process Environment;
msc RTOS_Trace;
Pr11: instancehead process Pr1;
Pr12: instancehead process Pr1;
Pr11: condition Idle;
Pr12: condition Idle;
Pr11: in Go,0 from MyExtTask3;
Signal Go received in Pr1:Instance1
dynpr14: instancehead process dynpr1;
Pr11 : create dynpr14;
Pr11: condition Wait;
Pr12: in Go,1 from MyExtTask3;
Signal Go received in Pr1:Instance2
dynpr15: instancehead process dynpr1;
Pr12 : create dynpr15;
Pr12: condition Wait;
dynpr14: set T1,2 (5000); /* #SDTNOW(269) */
dynpr14: condition wait;
dynpr15: set T1,3 (5000); /* #SDTNOW(276) */
dynpr15: condition wait;
dynpr14: timeout  T1,2; /* #SDTNOW(5284) */
dynpr14: out Terminating,4 to Pr11;
dynpr14: endinstance;
Pr11: in Terminating,4 from dynpr14;
Pr11: out Ok,5 to MyExtTask3;
Ok received in MyExtTask with paramer = 1
dynpr15: timeout  T1,3; /* #SDTNOW(5463) */
dynpr15: out Terminating,6 to Pr12;
dynpr15: endinstance;
Pr12: in Terminating,6 from dynpr15;
Pr12: out Ok,7 to MyExtTask3;
Ok received in MyExtTask with paramer = 2
```

Setting the XOS_TRACE flag should result in the following output:

```
** Process Pr1:9901455 created **

** Process Pr1:9901456 created **

** Process instance 9901455 **
Pr1: nextstate Idle
```

```
** Process instance 9901456 **
Pr1: nextstate Idle

** Process instance 9901455 **
Pr1: input signal Go
Signal Go received in Pr1:Instance1
Pr1: process dynpr1:9901458 created
Pr1: nextstate Wait

** Process instance 9901456 **
Pr1: input signal Go
Signal Go received in Pr1:Instance2

** Process instance 9901458 **

DynPr1: Set timer T1
Process instance 9901459
DynPr1: nextstate wait

** Process instance 9901458 **
DynPr1: input signal T1
DynPr1: signal Terminating sent
DynPr1: stopped

** Process instance 9901455 **
Pr1: input signal Terminating
Pr1: signal Ok sent
Pr1: dash nextstate
Ok received in MyExtTask with parameter = 1

** Process instance 99014516 **
DynPr1: input signal T1
DynPr1: signal Terminating sent
DynPr1: stopped

** Process instance 99014513 **
Pr1: input signal Terminating
Pr1: signal Ok sent
Pr1: dash nextstate
Ok received in MyExtTask with parameter = 2
```

Standard Model

In the standard model each instance of the Pr1 and the DynPr1 processes will be represented by an OS task (in all four tasks). The environment is represented by a task called `MyExtTask`. This task is external to the SDL system.

Instance Set Model

In the instance set model there will be two OS tasks, one each for Pr1 and DynPr1. The environment is represented by a task called `MyExTask`, just as in the standard model. This task is external to the SDL system.

How Signals are Sent to and from the Environment

There is an external task called `MyExtTask` which is written in C. It sends the signal Go into the SDL system and receives the signal Ok back by using services in the operating system.

The `HOOK_MAIN_AFTER_PROCESS_RELEASE` macro in `scttypes.h` is used to create `MyExtTask` as soon as the SDL system allowed to run.

The source code for the external task is placed in the file `MyExtTask.c`. This file is specific to the selected operating system because it calls the operating system directly.

Tight Integration Code Reference

This section explains data types, procedures and macros used in a Tight Integration (Light Integrations are explained in the Master Library).

General Macros

XPP(x)

The macro `XPP` is used in function declarations to specify the function parameters. It is defined like this:

```
#define XPP(x)  x
```

if function prototypes according to ANSI C can be used.

xptrint

The following type is also always defined:

```
#define xptrint unsigned
```

where `xptrint` should be an `int` type with the same size as a pointer.

xPrsNode and xPrdNode

```
typedef struct xPrsStruct  *xPrsNode;
typedef struct xPrdStruct  *xPrdNode;
```

`xPrsNode` and `xPrdNode` are pointers to structs holding instance data for an instance of a process or a procedure. Note that some parts of the structs are OS-dependent.

xInputAction, xNotInSignalSet ...

These defines specify the different ways of handling a signal.

Macros to Exclude Unnecessary Code

The following macros are defined to exclude unnecessary code for `IdNode` variables etc.:

```
#define XNOSTARTUPIDNODE
#define XOPTSIGPARA
#define XOPTDC
#define XOPTFPAR
#define XOPTSTRUCT
#define XOPTLIT
```

```
#define XOPTSORT
#define XNOUSEOFSERVICE
......
```

Macros to activate Signal-Free-Functions

The following macro must be defined to activate the Signal-Free-Functions. This is necessary if signals and timers with string parameter (dynamic allocated) are used - to avoid memory leaks.

```
#define XFREESIGNALFUNCS
```

If strings not are used as parameters in signals this flag should not be set cause it does lead to some performance deterioration. (Normally this define is set in the make-file).

For timer string parameters the define

```
#define XTIMERSWITHSTRINGPARAMS
```

must be set.

The following macro must be defined to activate the Signal-Free-Functions. This is necessary if signals and timers with string parameter (dynamic allocated) are used - to avoid memory leaks.

```
#define XFREESIGNALFUNCS
```

Default Priorities

One group of macros defines default priorities for processes and signals:

```
#ifndef xDefaultPrioProcess
#define xDefaultPrioProcess      RTOSPRIODEFAULT
#endif

#ifndef xDefaultPrioSignal
#define xDefaultPrioSignal       RTOSPRIODEFAULT
#endif
...
```

Macros to Implement SDL

First in this section is a macro defining the symbol table root:

```
xIdNode         xSymbolTableRoot;
```

XPROCESSDEF_C and XPROCESSDEF_H

These macros define the start-up function for a PAD function. It is this function that is called in the task creation. As mentioned before this function will after some variable initializations call the PAD function. The definition of the start-up function varies in different RTOS, that is why there is a second OS-specific macro here.

STARTUPSIGNAL, ALLOCPRSSIGNAL, etc.

Each signal in an application is assigned a unique integer value. The values 31992 through 32000 are reserved for internal signals like the start-up signal.

Variables in the PAD Function

PROCESS_VARS, PROCEDURE_VARS

These macros define the elements in the `xPrsStruct` and `xPrdStruct` respectively.

YPAD_YSVARP

This macro defines a variable representing a pointer to a signal's parameter area.

YPAD_YVARP

This macro defines the variable `yVarP` which represents the process instance data.

LOOP_LABEL, LOOP_LABEL_PRD, LOOP_LABEL_PRD_NOSTATE

These three macros define the eternal loop inside processes, procedures and procedures without a state.

START_STATE

Each state in a process and procedure is represented as an integer. The Start state will always have the value 0.

Using OSE Trace Features

Note:

IBM Rational has noticed that the OSE-trace feature can make the application crash in some situations. This seems to happen when a SDL process (OSE-task) sends a signal immediately before terminating. If you come across this problem, first check if the application works correctly when generated without OSE-trace.

Annex 1: Integration for OSE Delta

Introduction

This annex briefly describes the OSE Delta models and primitives used in the SDL Suite OSE tight integration. The presentation is focused on the differences from the OSE classic model described in the previous annex.

One section describes how to set up and run a simple test example in both a light and tight integration.

> **Note:**
> Third-party products referred to in this manual may have limitations that have impact on the usability of the SDL Suite. Please consult the supplier's support organization or the third-party product's technical reference documentation for up-to-date information about such limitations.

Principles

This integration is developed with OSE Delta Soft Kernel 3.2 and tested on a Sun workstation with SunOS Release 5.6.

The main differences between the OSE Delta and the OSE Classic model are:

- The OSE Delta model uses three semaphores to avoid synchronization problems in SDL start transitions.
- The timer is implemented in `systimer.c`, which is supplied by ENEA. This is not accurate and is only for demonstration purposes. You will have to supply a suitable timer implementation for the target environment.

Running the Test Example: Simple

Note:

The source file and examples for Tight Integrations are not included in the standard delivery. They are available as free downloads from IBM Rational Support web site.

Prerequisites

This test example is developed as an OSE Delta application on a Sun workstation. The makefile and compilation switches are set up for the application to run under an OSE Simulator for OS68. If you are using another configuration of OSE you probably need to edit the provided makefile.

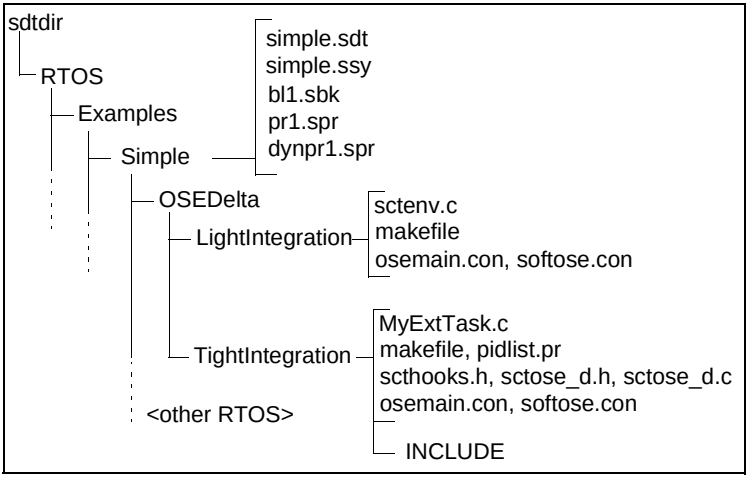


Figure 578: File structure for the Simple example

Light Integration

Limitations for the Light Integration

Please see the Release Guide.

Building a Light Integration

Please see the "Building and Running a Light Integration" on page 3359 for instructions.

Tight Integration

Limitations for the Tight Integration

Please read the Release Guide for details about limitations that apply to all systems using Tight Integration.

Building a Tight Integration

Please see the "Building and Running a Tight Integration" on page 3361 for instructions.

How Signals are Sent to and from the Environment.

The signal Go is sent from an external task `MyExtTask`. The code for this task is placed in the program file `MyExtTask.c`. This is the same as used for OSE Classic.

Annex 2: Integration for VxWorks

Introduction

This annex describes briefly the VxWorks models and primitives used in the SDL Suite VxWorks tight integration. The presentation is focused on the differences from the general model described earlier in this chapter.

One section describes how to set up and run a simple test example in both a light and tight integration.

> **Note:**
>
> Third-party products referred to in this manual may have limitations that have impact on the usability of the SDL Suite. Please consult the supplier's support organization or the third-party product's technical reference documentation for up-to-date information about such limitations.

Principles

This integration is developed with VxWorks Tornado 1.0 version and tested under VxSim version 5.3 on a Sun workstation with SunOS 4.1.4.

The main differences between VxWorks and the general model are:

- The VxWorks `msgQReceive` copies the Signal into a buffer when it is received. The sender makes free of the signal immediately after it has been sent and the receiver allocates a buffer (signal) before a receive statement.
- An extra optimization flag `XOPTSIGNALALLOC` has been introduced for the VxWorks tight integration. When freeing a signal’s memory, we place it in an availlist so that subsequent signal memory allocation calls can check to see if suitable sized memory already exists which may be reused. Otherwise memory is allocated as normal.

Running the Test Example: Simple

Note:

The source file and examples for Tight Integrations are not included in the standard delivery. They are available as free downloads from IBM Rational Support web site.

Prerequisites

This test example is developed as a VxWorks Tornado application on a Sun workstation. The makefile and compilation switches are set up for the application to run under an VxSim target simulator. If you are using another configuration of VxWorks you probably need to edit the provided makefile.

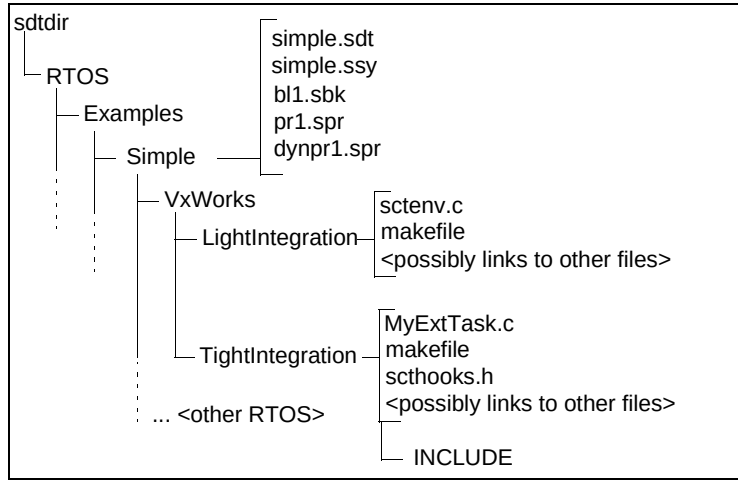


Figure 579: File structure for the Simple example

Note:

A VxWorks application is not allowed to contain a main function. The name of the generated main is changed to “root” with the compilation switch `-DXMAIN_NAME=root`.

Light Integration

Limitations for the Light Integration

Please see the Release Guide.

Building a Light Integration

Please see the "Building and Running a Light Integration" on page 3359 for instructions.

Tight Integration

Limitations for the Tight Integration

Please read the Release Guide for details about limitations that apply to all systems using Tight Integration.

Building a Tight Integration

Please see the "Building and Running a Tight Integration" on page 3361 for instructions.

Annex 3: Integration for Win32

This annex briefly describes integration with Win32. The presentation is focused on the differences from the general model described earlier in this chapter.

> **Note:**
>
> Third-party products referred to in this manual may have limitations that have impact on the usability of the SDL Suite. Please consult the supplier's support organization or the third-party product's technical reference documentation for up-to-date information about such limitations.

Principles

This integration is developed using the Microsoft 32-bit C/C++ Compiler Version 11.00.7022 and tested on NT 4.0, NT 3.51 and Windows 95 platforms.

The main differences between integration with Win32 and the general model are:

- Threads are created with the Win32 primitive CreateThread(). The thread is then automatically given an input queue the first time it calls a USER or GDI function
- xAlloc is implemented with Win32 function `HeapAlloc()`.
- xFree is implemented with Win32 function `HeapFree()`.
- The timer implementation uses the Win32 `GetTickCount()` function.

Running the Test Example: Simple

> **Note:**
>
> The source file and examples for Tight Integrations are not included in the standard delivery. They are available as free downloads from IBM Rational Support web site.

Prerequisites

This test example is developed as a Win32 console application on a PC. The makefiles and compilation switches are set up for the application to compile using the Microsoft compiler listed above.

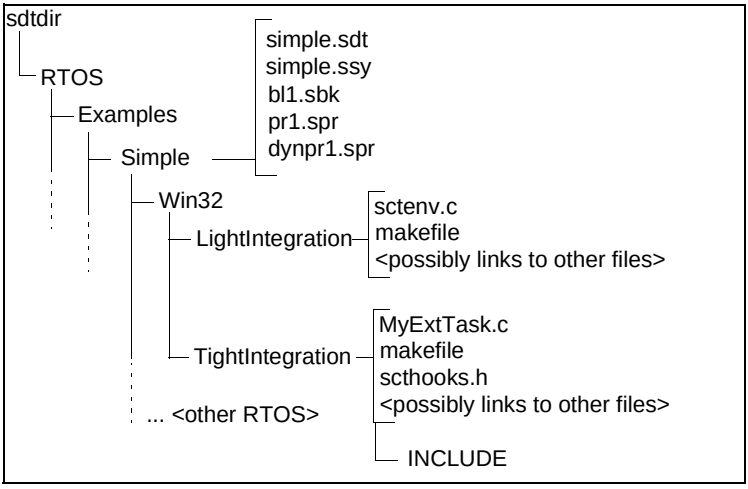


Figure 580: File structure for the Simple example

Light Integration

Limitations for the Light Integration

Please see the Release Guide.

Building a Light Integration

Please see the "Building and Running a Light Integration" on page 3359 for instructions.

Tight Integration

Limitations for the Tight Integration

Please read the Release Guide for details about limitations that apply to all systems using Tight Integration.

Building a Tight Integration

Please see the "Building and Running a Tight Integration" on page 3361 for instructions.

> **Note:**
> The command line length limitation for the Borland compiler can sometimes be exceeded. If this happens, you should define the `DEFINE MACROS` at the beginning of the `sctwin32.h` file.

Compiler Flags

The following defines (`#ifdef`) are used in this integration:

- `WIN32_INTEGRATION`: Ensures that the `sctwin32.h` file is included in each C file. Must be set in all cases.
- `XOS_TRACE`: Gives a textual trace for most of the SDL events by using printf to some device. This flag should not be used together with `XMSC_TRACE`.
- `XMSC_TRACE`: Will give a textual trace in the format of MSC/PR Z.120 by using printf. This trace is possible to view in the MSC Editor. This flag should not be used together with `XOS_TRACE`.
- `XMSC_EDITOR`: Used together with the `XMSC_TRACE` flag, the MSC trace is automatically displayed in the MSC Editor. Note that you must have the Organizer open on your machine.
- `X_ONE_TASK_PER_INSTANCE_SET`: States that the Instance Set Model is used. The Standard Model is otherwise chosen by default.
- `XERR`: When this flag is defined, the return status of all Win32 function calls will be printed.
- `XINCLUDE_HS_FILE`: Includes the system signal header file which is required for tight integrations. This file maps signal names to integers.

- XRTOSTIME: Should always be set for all tight integrations.
- XUSING_SCCD: This should be set when using the preprocessor SCCD to ensure that the windows header files are not included on the preprocessor pass. The files are included though on the compiler pass and this ensures that the preprocessed C files only contain the expanded the SDL Suite macros. It also helps greatly to speed up the process. Note that this flag only works with the Microsoft compiler and should not be used with any other compiler.
- XWINCE: This flag allows you to compile the integration for Microsoft WinCE target systems. This flag should not be used together with the MSC trace flags.

Annex 4: Integration for Solaris 2.6

Introduction

This annex describes briefly the Solaris 2.6 model and primitives used in the SDL Suite Solaris 2.6 tight integration. The presentation is focused on the differences from the general model described earlier in this chapter.

One section describes how to set up and run a simple test example for a tight integration.

Note:

The Solaris 2.6 tight integration is fully POSIX compliant. For this reason it will not work with earlier versions of Solaris.

Note:

Third-party products referred to in this manual may have limitations that have impact on the usability of the SDL Suite. Please consult the supplier's support organization or the third-party product's technical reference documentation for up-to-date information about such limitations.

Principles

This integration is developed using cc:WorkShop Compilers 4.2 with Solaris 2.6 running on a workstation.

The main differences between the Solaris 2.6 integration and the general model are:

- In the file `sctsolaris.h`, the macros which contain the Solaris 2.6 specific function calls are implemented. The file `sctsolaris.c` contains the Solaris 2.6 integration specific functions.
- The Solaris 2.6 integration is fully POSIX compliant. SDL processes are mapped to POSIX threads using the `pthread_create()` function and POSIX queues are created for each thread using `mq_open()`. The threads are suspended when its corresponding queue is empty.

Running the Test Example: Simple

Note:

The source file and examples for Tight Integrations are not included in the standard delivery. They are available as free downloads from IBM Rational Support web site.

Prerequisites

This test example is developed as a Solaris 2.6 application on a workstation. The makefile and compilation switches are set up for the application to run under Solaris 2.6 using the cc:WorkShop Compilers 4.2 compiler.

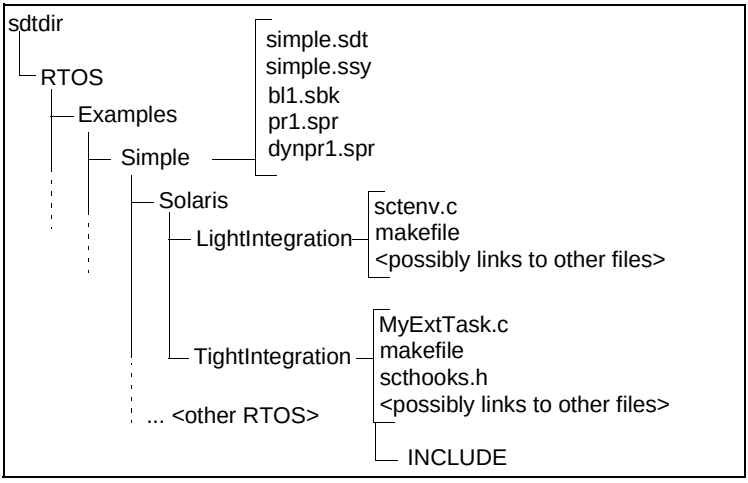


Figure 581: File Structure for the Simple example.

Light Integration

Limitations for the Light Integration

Please see the Release Guide.

Building a Light Integration

Please see the "Building and Running a Light Integration" on page 3359 for instructions.

Tight Integration

Limitations for the Tight Integration

Please read the Release Guide for details about limitations that apply to all systems using Tight Integration.

Building a Tight Integration

Please see the "Building and Running a Tight Integration" on page 3361 for instructions.

Compiler Flags

The following defines (`#ifdef`) are used in this integration:

- `SOLARIS_INTEGRATION`: Ensures that the `sctsolaris.h` file is included in each C file. Must be set in all cases.
- `XOS_TRACE`: Gives a textual trace for most of the SDL events by using printf to some device. This flag should not be used together with `XMSC_TRACE`.
- `XMSC_TRACE`: Will give a textual trace in the format of MSC/PR Z.120 by using printf. This trace is possible to view in the MSC Editor. This flag should not be used together with `XOS_TRACE`.
- `XMSC_EDITOR`: Used together with the `XMSC_TRACE` flag, the MSC trace is automatically displayed in the MSC Editor. Note that you must have the Organizer open on your machine.
- `X_ONE_TASK_PER_INSTANCE_SET`: Should be defined when the alternative runtime model is to be used.
- `XINCLUDE_HS_FILE`: Includes the system signal header file which is required for tight integrations. This file maps signal names to integers.
- `XRTOSTIME`: Should always be set for all tight integrations.

Annex 5: Generic POSIX Tight Integration

Introduction.

Note:

The Generic POSIX integration is based on the Solaris integration. For a description and instructions on how to generate and run the example see “Annex 4: Integration for Solaris 2.6” on page 3379.

Annex 6: Building a Threaded Integration

Introduction

This Tutorial, on how to create a Threaded integration, is developed on a Windows machine and is intended to be run under a Windows OS. If you want to use this example on another machine and for another OS, please remember to choose the appropriate integration and compiler for your OS in the Targeting Expert.

Preparations

The same SDL source files for the example Simple, that is used in the Light integration example will be used in this tutorial.

Copy the Source files for the Example: Simple

1. Create your own test directory and enter it.
2. Copy the SDL source files for the example Simple:

   ```
   cp <installation>/sdt/sdtdir/RTOS/Example/Sim-
   ple/*.s*
   ```

3. Copy the environment file from the Win32/ThreadedIntegration directory:

   ```
   cp <your Installation>/sdt/sdtdir/RTOS/Examples/Sim-
   ple/Win32/ThreadedIntegration/MyExtTask.c
   ```

4. Start the SDL Suite and open the system file for the Simple example.

Partition the system using the Deployment Editor

1. Create a new deployment diagram and call it Simple.
2. Edit the deployment diagram according to the figure below, see : Signalling in Threaded Integration.

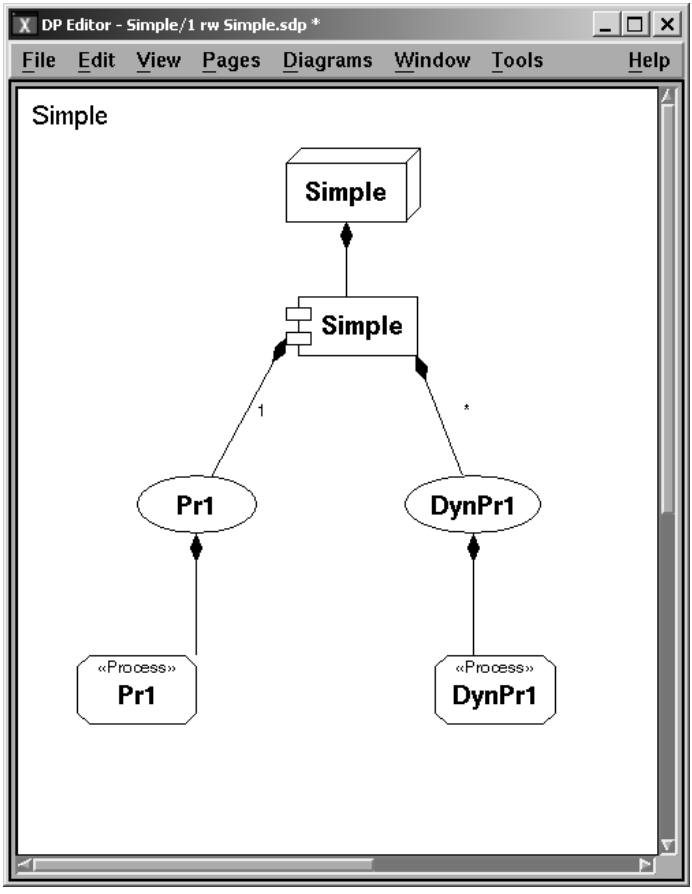


Figure 582 Deployment Diagram for the example Simple

3. Make sure that you got the multiplicity right on the aggregation line from the component to thread.

   To check this you can double click on the line. The right value should be:

   – 1 - On aggregation line from component to thread Pr1.

   – * - On aggregation line from component to thread DynPr1.

The multiplicity on the aggregation lines specifies how the component should be mapped to threads.

– A '*' means that each instance of the component should be mapped to an individual thread.

– A name means that the entire component should be mapped to a thread.

In our example this means that there should be one thread for **all** instances of Pr1 and one thread for **each** instance of DynPr1.

4. Double-click the component symbol. In the Symbol Details window specify that the integration model should be Threaded, see : Signalling in Threaded Integration.

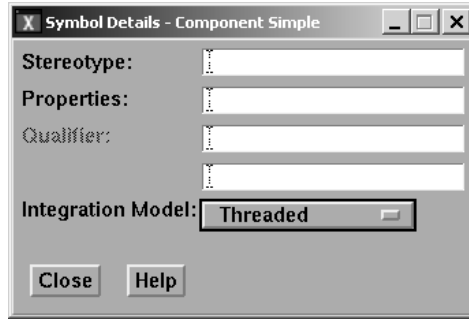


Figure 583 Symbol Details for the Component Simple

5. Double-click the thread symbol for Pr1. Specify the following Thread Parameters for the Thread P1:

   Thread Stack Size = 2048
   Thread Priority = 8
   Queue Size = 128
   Max Signal Size = 1024

6. Double-click the object symbol and specify that the stereotype should be Process and that the qualifier (for Pr1) should be:

   – Simple/Bl1/Pr1.

   Make the appropriate specifications for the object Symbol for DynPr1(Qualifier = Simple/Bl1/DynPr1).

7. Save the deployment diagram.
8. Select the deployment diagram in the Organizer and open the Targeting Expert from the Generate menu.
9. Choose the integration: Threaded Integrations->Win32 threaded.
10. You will be prompted if you want to generate the sdl_cfg.h file. Select No!
11. Disable the Generation of Environment Function by deselecting Environment Functions in the Environment section of the window.
12. Click on the Compiler/Linker/Make line in the Partitioning Diagram Model. You should now see the following in the Targeting Expert window, see:: The Compile/Linker/Make Window in Targeting Expert.

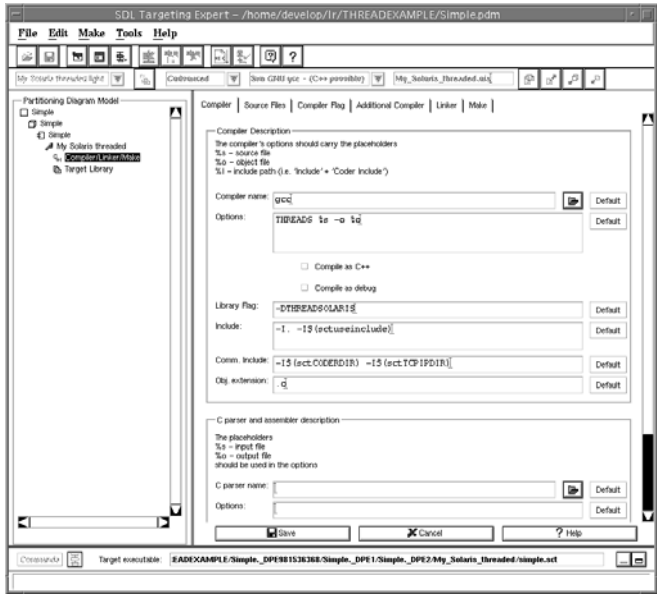


Figure 584: The Compile/Linker/Make Window in Targeting Expert.

13. Define the following flag: THREADED_SIMPLE_EXAMPLE in the Compiler description/Options window. This flag will start the External threads in the simple example.

14. Define the following Compilation flags:

    – THREADED_XTRACE,

    – THREADED_MSCTRACE

    by selecting the flags: *SDL trace and MSC* trace in the *Target library/Kernel window*

15. Add the `MyExtTask.c` file as a new source file.

    Click on the Source Files entry in the window and add the `MyExtTask.c` file in the source file list.

    Save the settings. You are now prompted again to generate configuration file, this time select Yes.

16. You should now be back in the Analyze/Generate code window and be ready to generate the application.

    Do a full Make and if you have followed the instructions the Targeting Expert will now analyze, generate code, generate makefile, compile and link the application.

17. Run the application simple.sct. Please note that you have to traverse down in the generated directory structure to find the application.

    You will find the application in a subdirectory similar to this path:

    – `<your test directory>/Simple._DPE981536368/Simple._DPE1/Simple._DPE2/Win32_threaded/....`

The output you should see when you run the application should be as follows:

```
Connected with the Postmaster.
*   OUTPUT of go to Pr1:1
*      Parameter(s) : 1
*** NEXTSTATE  Idle
*   OUTPUT of go to Pr1:2
*      Parameter(s) : 2
*** NEXTSTATE  Idle
Signal Go received in Pr1:Instance1
*   CREATE DynPr1:1
*** NEXTSTATE  Wait
Signal Go received in Pr1:Instance2
*   CREATE DynPr1:2
*   SET on timer t1 at 17.5580
*** NEXTSTATE  Wait_t1
*** NEXTSTATE  Wait
*   SET on timer t1 at 17.5880
*** NEXTSTATE  Wait_t1

*** TIMER signal was sent
*   Timer    : t1
*   Receiver : DynPr1:1
*** Now      : 17.5650
*   OUTPUT of t1 to DynPr1:1
*   PROCEDURE START Prd
Not doing much
*   PROCEDURE RETURN Prd
*   OUTPUT of terminating to Pr1:1

*** TIMER signal was sent
*   Timer    : t1
*   Receiver : DynPr1:2
*** Now      : 17.6050
*   OUTPUT of t1 to DynPr1:2
*** STOP   (no signals were discarded)
*   PROCEDURE START Prd
*   OUTPUT of ok to env:1
*      Parameter(s) : 1
Signal Ok received with the following parameter:1
*** NEXTSTATE  Wait
Not doing much
*   PROCEDURE RETURN Prd
*   OUTPUT of terminating to Pr1:2
*** STOP   (no signals were discarded)
*   OUTPUT of ok to env:1
*      Parameter(s) : 2
Signal Ok received with the following parameter:2
*** NEXTSTATE  Wait
```

Figure 585 Textual SDL trace for Simple example.

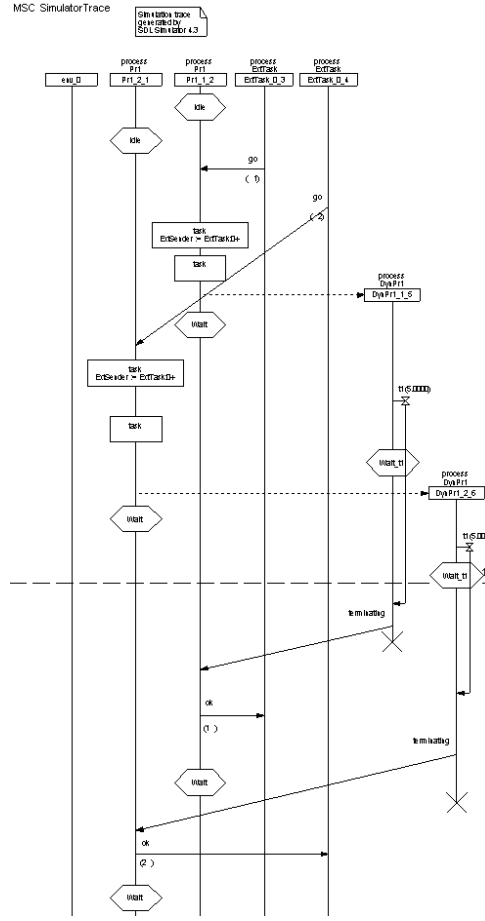


Figure 586 MSC trace for Simple Example

Chapter 64 Integration with Operating Systems

Chapter

65

The Cmicro SDL to C Compiler

The Cmicro SDL to C Compiler translates your SDL system into a C program that you can compile together with the Cmicro Library and the SDL Target Tester target library. The Cmicro Library and the SDL Target Tester target library is not available as a pre-linked library but is delivered as source to enable scaling of the kernel. The scaling is dependent upon the SDL system characteristics. This chapter is a reference manual for the Cmicro SDL to C Compiler.

- **In chapter 66, *The Cmicro Library*, you will find information about how to customize your own libraries for specific purpose, such as application generation for target computers. The chapter also describes the structure of the generated C code and the internal data structures in the generated C code. The Cmicro Library is only of use when compiled with the code which is generated by the Cmicro SDL to C Compiler.**
- **In chapter 67, *The SDL Target Tester*, you will find a reference to the features, which enables testing in a host-target environment. The SDL Target Tester is applicable for the Cmicro SDL to C Compiler and the Cmicro Library only.**

Application Area for the Cmicro SDL to C Compiler

The application area for the Cmicro SDL to C Compiler is:

- Generation of applications, including embedded system applications with real time characteristics (Configuration: Cmicro Library and generated C code running on target).
- Generation of target debug applications, including embedded system applications with real time characteristics (Configuration: Cmicro Library, generated C code and SDL Target Tester running on target).

In this part of the chapter, the general behavior of the Cmicro SDL to C Compiler, as seen from the users point of view, is discussed.

Highly Optimized Code for Target

The generated code in combination with the Cmicro Library is highly optimized, which is unavoidable for microcontrollers and real-time applications. Some optimizations have been possible only by introducing restrictions in the use of SDL. Other optimizations have been possible by generating more compact code. For the restrictions in the use of SDL please see "SDL Restrictions" on page 3450. Details regarding the output of the Cmicro code generation can be found in "Output of Code Generation" on page 3418.

Target Debug

With the generated code it is possible to debug the application on the target using the Cmicro Library and the SDL Target Tester library. The parts of the Cmicro code generation which are used for the SDL Target Tester are also highly optimized. Please see chapter 67, *The SDL Target Tester*.

Overview of the Cmicro SDL to C Compiler

The SDL Analyzer, which can be invoked from the Organizer, contains an SDL parser, an SDL semantic checker, and – among other code generators – the Cmicro SDL to C Compiler.

Many options can be chosen from the user which affect the analysis of the SDL system. Furthermore, a lot of error checks are performed automatically before code generation starts. This makes it possible to improve written SDL specifications before any run-time testing must be done.

The options that the user may choose for analysis and the error checks that are performed by the analyzer are described in chapter 54, *The SDL Analyzer*.

At some places the Cmicro SDL to C Compiler can be used in exactly the same way as the Cadvanced/Cbasic SDL to C Compiler can be used. At some other places the use of this C Code Generator, or what this C Code Generator produce, is different.

The Cmicro SDL to C Compiler generally can process the same input as the Cadvanced/Cbasic SDL to C Compiler can. The differences are explained within this chapter.

The differences in the output of the both code generators are described within the subsection "Output of Code Generation" on page 3418.

The overall differences of the both code generators are described in the section "Differences between Cmicro and Cadvanced" on page 3456 in chapter 66, *The Cmicro Library*.

The following subsections describe how the Cmicro SDL to C Compiler might be used.

Generated Files

The C files, which are generated by the Cmicro SDL to C Compiler, can only be used in connection with the Cmicro Library and the SDL Target Tester. It is not possible to validate and simulate the SDL system with the C code generated by Cmicro as this code is only suitable for target applications. To simulate and validate the SDL system within the SDL Suite, the user has to choose the Cbasic SDL to C Compiler. In order to view the process of generating C applications see the Organizer's *Make* dialog in chapter 2, *The Organizer*.

The SDL Analyzer, which contains the Cmicro SDL to C Compiler can also be started as a stand-alone tool. For more information about this possibility please see *chapter 54, The SDL Analyzer*.

There are several steps that must be carried out before the generated C files can be compiled and linked together with the Cmicro Library. The user should follow the procedures that are documented in the section "Targeting using the Cmicro Package" on page 3481 in chapter 66, *The Cmicro Library*.

In the following subsections the different files that are generated are explained.

Generated Configuration File

The first file that is generated from the Cmicro SDL to C Compiler is called `sdl_cfg.h`. It is used to scale the Cmicro Kernel depending on what characteristics the SDL system has. This is called automatic scaling and automatic dimensioning facility.

The file contains a header, process ID declarations, and then a `#define` or a `/*NOT define ... */` for each of the flags that the Cmicro SDL to C Compiler can generate automatically.

Example 571: The Header of an sdl_cfg.h ----------

```
/* Program generated by SDL Suite.Cmicro <version> <date> */

#ifndef XSCT_CMICRO
#define XSCT_CMICRO
#endif

/* "sdl_cfg.h" file generated for system <systemname> */

#define XMK_CFG_TIME <GenerationTime>
```

The XMK_CFG_TIME macro is used internally when compiling and executing with the SDL Target Tester takes place. With this macro, a rough consistency check for the generated files is done. The <GenerationTime> of the different files that are generated is compared in the Library and by the SDL Target Tester. If there is an inconsistency, compilation errors will occur.

The rest of the file sdl_cfg.h is about the automatic scaling and automatic dimensioning of the SDL system. It may look for example like:

Example 572: The Tail of an sdl_cfg.h ----------

```
#define MAX_SDL_PROCESS_TYPES <N>
#define XMK_USED_ONLY_X_1
#define MAX_SDL_TIMER_TYPES <X>
#define MAX_SDL_TIMER_INSTS <Z>
#define XMK_HIGHEST_SIGNAL_NR 4
/* NOT #define XMK_USED_TIMER */
/* NOT #define XMK_USED_DYNAMIC_CREATE */
/* NOT #define XMK_USED_DYNAMIC_STOP */
/* NOT #define XMK_USED_SAVE */
#define XMK_USED_SIGNAL_WITH_PARAMS
/* NOT #define XMK_USED_TIMER_WITH_PARAMS */
/* NOT #define XMK_USED_SENDER */
/* NOT #define XMK_USED_OFFSPRING */
/* NOT #define XMK_USED_PARENT */
/* NOT #define XMK_USED_SELF */
/* NOT #define XMK_USED_PWOS */
/* NOT #define XMK_USED_INITFUNC */
```

For a first rough understanding of the meaning of the different flags: The SDL system from above contains <N> process types (using SDL'88 terminology), all the processes are declared in the form (0,1) or (1,1). There are <X> timers declared (in this case, <X> must be 0, because XMK_USED_TIMER is undefined, and the system uses an amount of <Z> signals. The system does not use any create or stop (XMK_USED_DYNAMIC_CREATE and XMK_USED_DYNAMIC_STOP are undefined). In this way all the other flags have special meaning.

For explanations about the different flags the user should refer to "Automatic Scaling Included in Cmicro" on page 3519 in chapter 66, *The Cmicro Library*.

Generated C File

Assumed, that the user selected *"No separation"* in the Organizer's Make Dialog, and no partitioning is used, then the Cmicro SDL to C Compiler will generate one C file per SDL system. This file contains all the characteristics of the SDL system including all the declarations that

the SDL system itself needs. For an explanation of this file see "Output of Code Generation" on page 3418.

Generated Environment Header File

There is one file generated from the Cmicro SDL to C Compiler that contains all the definitions and declarations that are necessary to implement the environment functions `xInEnv` and `xOutEnv`.

The file is generated only if the option *Environment header file* in the Targeting Experts is switched on.

The file is called `<systemname>.ifc` and it contains a header, the type definitions used on system level (newtypes, syntypes, synonyms), the signal IDs and the structure type definitions for the parameters of the signals.

The `IFC_FILENAME` macro is defined when generating code from the Make dialog, i.e. when using the comp.opt and make.opt files. The macro is not automatically defined when using Targeting Expert. The macro contains the full path to the ifc file, even if that file has been chosen not to be generated.

Typical usage of this macro is to include the ifc file when implementing the environment functions or whenever you would like to include the ifc file in external code.

Example 573: Compiler options macro when using Targeting Expert–

```
-DIFC_FILENAME=
"'C:\MyProject\MySys._0\Application_CA\MySys.ifc'"
```

If Targeting Expert is used and you would like to use the `IFC_FILENAME`, you must define the macro by yourself in the compiler options.

Example 574: The Header of an <systemname>.ifc file–

```
#ifndef X_IFC_z_env01
#define X_IFC_z_env01
#define XMK_IFC_TIME <GenerationTime>
```

The `XMK_IFC_TIME` macro is used internally when compiling and executing with the SDL Target Tester takes place. With this macro, a rough consistency check for the generated files is done. The

<GenerationTime> of the different files that are generated is compared in the Library and by the SDL Target Tester. If there is an inconsistency, compilation errors will occur.

Caution!

As there are defines generated that contain no prefixes, there might be compiler warnings like `Illegal redefinition of macro`. Such redefinitions should never be ignored because fatal errors during run-time may occur. The user should introduce a prefix for signals or sorts with different meaning on SDL level, in order to map these names to unambiguous identifiers in C.

More explanation about the environment header file is given in chapter 66, *The Cmicro Library*.

Sorts

Followed by the header the section about sorts follows. The sorts are generated according to the documentation in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*.

Signal IDs and Parameter Structures

Next, the definitions for signals follows, which consists of:

- A C comment with a comment explaining if the signal is `IN` or `OUT` as seen from SDL
- An optional declaration of a C structure type definition (if the signal carries parameters)
- The definition of the signal ID

For easy interpretation: For an SDL signal **without** parameters, going from the environment to SDL, like:

```
signal SIn;
```

the following is generated into the `.ifc` file:

```
/* SIn IN */
#define SIn  <X>
```

For an SDL signal **with** parameters, going from SDL to the environment, like

```
signal SOut (integer, mystruct, boolean);
```

the following is generated into the `.ifc` file:

```
/* SOut OUT */
typedef struct {
    SIGNAL_VARS
    SDL_Integer  Param1;
    mystruct     Param2;
              /* mystruct declared in the section */
              /* declaring sorts                  */
    SDL_Boolean  Param3;
} yPDef_<UniquePrefix>_SOut;

typedef yPDef_<UniquePrefix>_SOut  *yPDP_<UniquePrefix>_SOut;
#define yPDP_SOut                  yPDP_<UniquePrefix>_SOut
#define yPDef_SOut                  yPDef_<UniquePrefix>_SOut
#define SOut                       <X>
```

At least the signal ID (here: `SOut`) and the name of the structure (here: `yPDef_SOut`) must be used in the `xOutEnv` C function in this case.

The code generation of structure types and signal IDs is (except the C comment about `IN` or `OUT`) independent from the direction the signal goes.

Process IDs

At last the process ID declarations are generated as `#define` values in C, like:

```
#define XPTID_<auto-prefix>_MyProcess 0
```

where the first process in the system is the value of 0 assigned, the second process gets the value 1, and so on. Due to the implementation of SDL'92 object orientation in the Cmicro SDL to C Compiler, there is also an automatic prefix generated. Using this prefix in the user's environment functions, it is possible to distinguish between several processes with the same name. Please refer to "Generation of Identifiers" on page 3446 for more information.

Generated Make File

The Cmicro SDL to C Compiler generates a file that contains production rules for the C program. This file can be used together with "make" facility only. The file is called `<systemname>.m`.

Additionally there is an ASCII file called `<systemname>_gen.m` which gives a list of all the generated files. This files is used by the Targeting Expert to generate a makefile. Please see "Generated Makefile" on page 2997 in chapter 59, *The Targeting Expert*.

Generated Symbol File

The generated symbol file is used to store symbolic information about the SDL system. The file has meaning for the host part of the SDL Target Tester only and is called `<systemname>.sym`. It is used for SDL Target Tester purposes only and is described within "The Host Symbol Table" on page 3731 in chapter 67, *The SDL Target Tester*.

Generated Kernel Group File

The generated kernel group file contains information about process names. This file is especially used in integrations, when OO is used and processes are instantiated. Using the information from this file, it is possible to distinguish between several process instantiations with the same name.

Implementation

In this section the implementation details are discussed. These details are meaningful for understanding how a generated Cmicro application does work.

Time

For host simulation, with the predefined integration settings, a time unit represents one second. In target applications, time is to be implemented by the user (see subsection "Defining the SDL System Time Functions in mk_stim.c" on page 3528 in chapter 66, *The Cmicro Library*).

Real Time

If real time is used, then there will be a connection between the clock in the executing program and the wall clock. For applications the user must provide the connection with the wall clock, normally the hardware timer.

> **Note:**
>
> The C standard function `time` used as the real time clock returns the time in seconds. The implementation of the clock can be changed by re-implementing the function `xmk_NOW` in `mk_stim.c`.

Scheduling

The Cmicro Kernel does not use a process ready queue. It processes the signals in the order of their appearance. To do this, there is a signal queue which stores the signals sent to any process (either internally or externally). There are different ways to influence the scheduling when using the Cmicro SDL to C Compiler:

- assigning priorities to processes
- assigning priorities to signals
- any combination of process and signal priorities

Assigning Priorities to Processes – Preemptive Scheduling

It is possible to assign priorities to process types (using SDL'88 terminology). The processes' priorities are assigned when designing the SDL system. They are assigned using the #PRIO directive.

There are some things to be kept in mind when using process priorities:

- Priorities have to begin with zero.
- Priorities have to be consecutive.
- All instances of a type have the same priority (SDL'88 terminology).
- Priority decreases with increasing numbers (zero is the highest priority level).
- The default priority is to be in the range of zero to the lowest priority number.

The Cmicro Kernel handles process priorities by collecting all signals sent to processes of the same priority in a separate queue. Thus, there is a queue for each priority level.

While the SDL system is running the kernel checks for signals in the queues with decreasing priority. This check takes place whenever an SDL output appears or a process performs an SDL nextstate operation. Because of the kernel checking for signals whenever an output takes place, it is possible to have preemptive scheduling.

Assume, there are two process types lowprio and highprio. Let process type lowprio have the priority one and process type highprio have the priority zero.

If an instance of process type lowprio performs an output to process type highprio, there appears a signal in a queue of a higher priority level (zero is the highest priority level available, process lowprio has priority one) which leads to the kernel immediately working on the signal sent to the process highprio. The transition of process lowprio will not end until process highprio has finished its transition invoked by the signal.

This way of scheduling is implemented using recursion.

Note:

Process priorities are available only when using a compiler which can handle recursion.

There is basically no restriction on the number of priority levels, but the target and compiler used will of course limit the depth of recursion.

As a general recommendation process priorities should not be assigned one per process type, but the process types should be grouped according to their purposes and these groups should then be assigned a priority level.

Assigning Priorities to Signals

The signals in the queue(s) are normally ordered according to their appearance (FIFO-strategy). By assigning priorities to signals this ordering is user definable. The directive #PRIO is used to assign a priority to signals.

Priority increases with decreasing numbers, but there is no restriction to use consecutive numbering.

Whenever a signal is sent, it is inserted into the signal queue(s) according to its priority.

Assume, there is a process performing two signal outputs, first_sig and second_sig. Using the standard FIFO-strategy signal first_sig would be worked on before signal second_sig. But with signal priorities and signal first_sig assigned priority fifty and signal second_sig assigned priority twenty, signal second_sig would be in front of signal first_sig in the queue and thus would be worked on before signal first_sig.

For more details please refer to "Assigning Priorities – Directive #PRIO" on page 3415.

Combinations of Signal/Process Priority

Every combination of signal and process priorities may be used. In this way it is possible to adapt the scheduling to the users' needs.

Note:

Without process priorities a transition once started will have to be finished before the next transition can be dealt with. This is valid regardless of the time it will need to finish a transition.

Synonyms

External Synonyms

External synonyms can be used to parameterize an SDL system and thereby also a generated program. The values that should be used for the external synonyms must be included as macro definitions into the generated code, for instance by including another header file.

Using a Macro Definition

To use a macro definition in C to specify the value of an external synonym, the user should perform the following steps:

1. Write the actual macro definitions on a file.

   Example 575: Macro Definition

   ```
   #define synonym1 value1
   #define synonym2 value2
   ```

   The synonym names are the SDL names (without any prefixes).

2. Introduce the following `#CODE` directive at the system level among the SDL definitions of synonyms, sorts, and signals, for example, but before any use of the synonyms.

   Example 576: #CODE Directive

   ```
   /*#CODE
   #TYPE
   #include "filename"
   */
   ```

If this structure is used, the value of an external synonym can be changed merely by changing the corresponding macro definition and recompiling the system.

Procedure Calls and Operator Calls

In SDL-92, value returning procedures and operator calls are introduced. This means, that an SDL procedure can be called within an expression. As the Cmicro SDL to C Compiler cannot handle procedures with states, it is not necessary to map such calls to a different scheme.

Example 577: Procedure Call

```
TASK i := (call p(1)) + (call Q(i,k));
```

is translated to something like:

```
i = p(1) + Q(i,k);
```

Note:

The value returning procedure calls are transformed to C functions which return values.

Operators which are defined using operator diagrams, are as in the models in the SDL recommendation, treated exactly as value returning procedures.

Generation of PAD function

The code generation for the PAD function is different compared with Cadvanced, in the way that code that is common in process types is copied into the PAD function for instantiated processes. This is implemented in contrast to Cadvanced, where for each process type definition there is a C function generated once, that is called by the instantiated PAD function, for common code. This makes a difference when system partitioning and/or file separation is used.

Any

‘Any’ should not be used in applications using the Cmicro SDL to C Compiler, as it leads to an error message.

Calculation of Receiver in Outputs

The Cmicro SDL to C Compiler is a code generator using the semantics of SDL-92 with some restrictions. The behavior for output is according to the rules described in the following:

- For an output without TO and without VIA in SDL, the Cmicro SDL to C Compiler calculates the receiver of the signal during code generation. If there is more than one possible receiving process type, then an error message will be printed out.
- For an output without TO and without VIA in SDL, it is also possible to have one process type, but more than one receiving instance of the signal. The response is that any of the living possible receivers may be selected during execution time. If no receiver is found, the C function `ErrorHandler` will be called.
- For an output with the VIA clause, the behavior of the Cmicro SDL to C Compiler is in principle the same as for an output without TO It computes the possible receivers in an output with the VIA clause and if there are several possible receivers, an error message is produced. The only difference between output with VIA and output without TO is that VIA can restrict the amount of possible processes.
- If output with TO is used in the above cases, no ambiguity can occur. The addressing of the process is then performed by a run-time variable.
- The possibility of specifying the name of a process when using OUTPUT TO is implemented. This is an SDL-92 feature. The Cmicro SDL to C Compiler behaves in the same way as when using implicit addressing (output without to).
- The broadcast feature of SDL-92 (VIA ALL) is not implemented, because it is not a real broadcast and not very useful for Cmicro Applications.

Abstract Data Types

In this section the specialities and exceptions about abstract data types for Cmicro are discussed only. A complete documentation about the abstract data types is given in chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*.

General C Definitions

All the macros and external definitions for functions can be found in the file `sctpred.h` except for the PId sort which is handled in the file `ml_typ.h`.

The C functions for the handling of predefined sorts are defined in the file `sctpred.c`.

On UNIX these files can be found in `$sdtdir/cmicro/kernel`.

In Windows these files can be found in the installation under `%SDTDIR%\cmicro\kernel`.

Exceptions for SDL Predefined Types

A general exception existing for all the predefined types is that the user must configure which predefined types are to be compiled into the target C program. This is necessary to hold the target C program as small as possible. The configuration is to be performed with the help of the Targeting Expert, please view "Configure and Scale the Target Library" on page 2946 in chapter 59, *The Targeting Expert*.

Caution!

Problems will occur during compilation when the configuration is not according to what the SDL system needs. The user should refer to the explanations about manual scaling in chapter 66, *The Cmicro Library*.

External Synonyms

External synonyms are to be defined by the user in the following way.

For a synonym like

```
synonym xternal integer = EXTERNAL;
```

Cmicro expects to see `xternal` as a `#define` value that is to be defined by the user. This can be done for example in the following way:

```
synonym xternal integer = EXTERNAL;
/*#CODE
#TYPE
#ifdef XSCT_CMICRO
#define xternal 7
#endif
*/
```

This also means, that if `xternal` is not defined from the user, it will lead to compilation errors.

Charstring

Charstrings can be used either in the usual way as they are when using Cadvanced, or they can be used in a restricted way. The decision is up to the user and is a question of configuration. The user should be aware that some of the predefined sorts from ASN.1 are based on the implementation of SDL charstrings. This is discussed in subsection "Support of SDL Constructs" on page 3510 in chapter 66, *The Cmicro Library*.

Time/Duration

The predefined data types Time and Duration are implemented in a more or less restrictive way. It is possible to specify a real value for Time and Duration on SDL level, like 23.45. The Cmicro Library uses only the integer part in front of the dot, 23 in this example. The mapping of SDL time units to time units in a target application is – in any case – up to the user.

UnionC

The #UNIONC directive is not recommended when using the Cmicro SDL to C Compiler because there is no support for checking the validity of the component selection. Both the #UNION directive and the CHOICE concept are a better alternative.

Predefined Generators Array, String, Powerset, Bag, Ref

These generators are implemented in Cmicro, but the user should be aware that the use of any of them requires that dynamic memory allocation is used in the target system. Generally, Cmicro tries to prevent the use of dynamic memory allocation whenever possible. The reasons for this are explained in chapter 66, *The Cmicro Library*.

ctypes.sdl

This package can be used together with Cmicro with the following restriction.

There are two operators that are excluded when Cmicro C code is compiled. The operators are "`CStar2CString`" and "`CharStar`".

The reason for this is that with Cmicro it is possible to define an array of char in C instead of the predefined solution of Cadvanced (to use dynamic memory allocation). This is discussed in subsection "Support of SDL Constructs" on page 3510 in chapter 66, *The Cmicro Library*.

byte.pr

This ADT can be used together with Cmicro in the same way as described for Cadvanced.

file.pr

This ADT is not useful for typical Cmicro applications (embedded systems usually do not provide a hard disk in Cmicro applications) and for that reason never has been tested. The ADT may however work with Cmicro.

idnode.pr

This ADT **cannot** be used together with Cmicro because it refers to Cadvanced code.

list1/list2.pr

This ADT **cannot** be used together with Cmicro because it refers to Cadvanced code.

long_int.pr

This ADT can be used together with Cmicro.

pidlist.pr

This ADT **cannot** be used together with Cmicro, generally.

Instead of `pidlist.pr`, the user may include the `cm_pidlist.pr` file. The use of this ADT is however restricted, because Cmicro implements a different scheduling algorithm. This means, that systems that

are successfully simulated first, may contain problems when a Cmicro target application is build and executed.

It is therefore recommended **not** to use neither `pidlist.pr` nor `cm_pidlist.pr`, in order to achieve the best possible SDL conformity.

random.pr

This ADT **cannot** be used together with Cmicro because it refers to Cadvanced code.

unsigned.pr

This ADT can be used together with Cmicro.

unsigned_long.pr

This ADT can be used together with Cmicro.

Default Values

Default values are in principle generated in the same way as with the Cadvanced SDL to C Compiler. It is however possible to configure the default value setting, which is explained in chapter 66, *The Cmicro Library*. The right configuration is essential to prevent illegal behavior.

Exceptions for Implementations of Operators

Read and Write Functions

The Cmicro SDL to C Compiler does not provide read and write functions. The reason is, that the Cmicro SDL to C Compiler mainly is used to build target applications, and not simulations. This is also a consequence of optimizing the target program. If the user uses the Q (question) operator, the Cmicro SDL to C Compiler ignores this.

Error Situations in Operators

In the C function used to implement operators (and literals), it is possible to define error situations and handle them as ordinary SDL run-time errors. The C library function `ErrorHandler`, with the following prototype

```
extern void ErrorHandler( xmk_OPT_INT errnum )
```

can be used for this purpose. `xmk_OPT_INT` is defined in `ml_typ.h`, normally as an ordinary C `int`. `errnum` may be one of the free values of error numbers. Please inspect `ml_err.h` in order to get a list of reserved values.

Example 578: Error Handler in Operator

```
  if ( strlen(C) <= 1 ) {
#ifdef XMK_USE_ERR_CHECK
    ErrorHandler (ERR_N_InvalidStringLength);
#endif
    return SDL_NUL;
  } else
    return C[1];
```

This is a simplified version of the test in the function for the operator First in the sort Charstring. Here the error situation is when we try to access the first character in a charstring of length 0. In this case the C function `ErrorHandler` is called and a default value is returned (NULL). By including the call to `ErrorHandler` between `#ifdef XMK_USE_ERR_CHECK - #endif` the function is only called to report the error, if error checks are turned on. The one parameter to the C function `ErrorHandler` should identify the error. The number must be given by the user.

Another possibility to route error messages to the host system is to use the C function `xmk_PrintString` of the SDL Target Tester, defined as:

```
extern void xmk_PrintString( char * )
```

Example 579: Error Handler in Operator

```
  if ( strlen(C) <= 1 ) {
#ifdef XMK_ADD_MICRO_TESTER
    xmk_PrintString ("ERR:Invalid Stringlength");
#endif
    return SDL_NUL;
  } else
    return C[1];
```

Access to Predefined Sorts based on Charstring

As already mentioned in earlier subsections, the user should be aware that some of the ASN.1 predefined sorts are based on the implementation of SDL charstrings. The user should also refer to subsection "Sup-

port of SDL Constructs" on page 3510 in chapter 66, *The Cmicro Library*.

To avoid problems one should be aware that Charstring is implemented as char * in C and take the consequences thereof. There are a number of help functions (that implement the operators for the Charstring sort) supplied in the run-time library that might be helpful when handling Charstrings.

It is usually necessary to allocate dynamic memory when an operator returning a charstring value is implemented. There are two help functions that should be used in connection with allocation and de-allocation of dynamic memory. These are documented in "Dynamic Memory Allocation" on page 3543 in chapter 66, *The Cmicro Library*.

Caution!

Do not use Charstring in SDL if you want to get a correct trace output with the SDL Target Tester, or if you want to use the Cmicro Recorder. In the last case, the use of charstring may lead to a fatal error when an SDL session is replayed.

Exceptions for Directives

Selecting File Structure for Generated Code – Directive #SEPARATE

The purpose of the separate generation feature is to specify the file structure of the generated program. Both the division of the system into a number of files and the actual file names can be specified. There are two ways this information can be given.

- Normally this information is set up in the Organizer, using the command in chapter 2, *The Organizer*. Here file names for the generated files can also be specified. In the *Make* dialog in the Organizer (see "Generated Files" on page 3394) it is possible to select full separate generation, user-defined separate generation, or no separate generation.
- For an SDL/PR file that is generated by running the SDL Analyzer as a stand-alone tool, the same information can be entered by

#SEPARATE directives directly introduced in the SDL program. Full separate file generation, user-defined separate file generation, or no separate file generation can be set up in the command interface of a stand-alone Analyzer, see *"Set-Modularity" on page 2491 in chapter 54, The SDL Analyzer*.

The Cmicro SDL to C Compiler can generate a separate file for:

- System (always separate)
- Block
- Process
- Procedure

Note:

Instantiations cannot be separated. If #SEPARATE directives are used, they should be placed directly after the first semicolon in the system, block, process, or procedure heading; see the following example.

Example 580: #SEPARATE Directive ––––––––––––––––––––––––

```
system S; /*#SEPARATE 'filename' */
block B; /*#SEPARATE */
process type P1 inherits PType; /*#SEPARATE */
process P2 (1, ); /*#SEPARATE */
procedure Q; /*#SEPARATE */
```

––

In the example above the two versions of separate directive, with or without file name, are shown. As can be seen a file name should be enclosed between quotes. The Cmicro SDL to C Compiler will append appropriate extensions to this name when it generates code.

If no file name is given in the directive, the name of the system, block, process, or procedure will be used to obtain a file name. In such a case the file name becomes the name of the unit with the appropriate extension (`.c .h`) depending on contents. The file name is stripped of characters that are not letters, digits or underscores.

The possibility to set up full, user-defined, or no separation in the Organizer's *Make* dialog and in the user-interface of a stand-alone Analyzer (see "Generated Files" on page 3394), can be used in a simple manner

to select certain default separation schemes. This setting will be interpreted in the following way:

- *No* separation.
  The whole system will be generated into one file.
- *User defined* separation.
  The system, each package, and each unit that the user has specified as separate will become a separate file.
- *Full* separation.
  The system, each package, each block, block type, and process, and process type will become a separate file. Note that even in this case a procedure is separate only if the user has specified it as separate.

Independently if *No*, *User defined*, or *Full* separation has been selected, the Cmicro SDL to C Compiler will use the file name specified in the *Edit Separation* dialog or the #SEPARATE directive, for a file that is to be generated.

An Example of the Usage of the Separate Feature

In the following example a system structure and the #SEPARATE directives are given. The same information can easily be set up in the Organizer as well. This example is then used to show the generated file structure depending on selected generation options.

Example 581: #SEPARATE Directive ––––––––––––––––––––––––

```
system S; /*#SEPARATE 'Sfile' */
  block B1; /*#SEPARATE */
    process P11; /*#SEPARATE 'P11file' */
    process P12;
  block B2;
    process P21;
    process P22; /*#SEPARATE */
```

––

Applying Full Separate Generation

If *Full* separate generation is selected then the following files will be generated:

Sfile.c	Sfile.h
B1.c	B1.h

P11file.c	
P12.c	
B2.c	B2.h
P21.c	
P22.c	

The .c files contain the C code for the corresponding SDL unit and the .h files contain the module interfaces.

Applying Separate Generation

If *User defined* separate generation is selected then the following files will be generated:

Sfile.c	Sfile.h	Contains code for units S, B2, P21
B1.c	B1.h	Contains code for units B1, P12
P11file.c		Contains code for unit P11
P22.c		Contains code for unit P22

The user defined separate generation option thus makes it possible for a user to completely decide the file structure for the generated code. The comments on files and extensions given above are, of course, also valid in this case.

Applying No Separate Generation

If the separation option *No* is selected, only the following file will be generated:

Sfile.c		Contains code for all units

The comments on files and extensions earlier are valid even here.

Guidelines

Generally a system should be divided into manageable pieces of code That is, for a **large system**, *full separate generation* should be used, while for a **small system**, *no separate generation* ought to be used. The possibility to regenerate and re-compile only parts of a system usually

compensate for the overhead in generating and compiling several files for a large system.

> **Note:**
> A file name has to be specified, using the Organizer *Edit Separation* command or the #SEPARATE directive, if two units in the system have the same name in SDL and should both be generated on separate files. Otherwise the same file name will be used for both units.

Assigning Priorities – Directive #PRIO

#PRIO for Processes

Priorities can be assigned to processes using the directive #PRIO. The process priorities will affect the scheduling of processes, see "Scheduling" on page 3469. A priority is a positive integer, where low value means high priority. #PRIO directives should be placed directly after the process heading in the definition of the current process.

Example 582: #PRIO Directive

```
Process P1; /*#PRIO 0 */
Process P2(1,1); /*#PRIO 1 */

Process P3 : P3Type; /*#PRIO 0 */
Process P4(1,1) : P4Type; /*#PRIO 1 */
```

Processes that do not contain any priority directive will have a user defined default priority with the name `xDefaultPrioProcess`.

There are some things to be kept in mind when using process priorities:

- Priorities have to begin with zero.
- Priorities have to be consecutive (0,1,2,3,4,5).
- All instances of a type have the same priority.
- Priority decreases with increasing numbers (zero is the highest priority level).
- The default priority is to be in the range of zero to the highest priority number, that is 0 or 1 in the example above.

#PRIO for Signals

Priorities can be assigned to signals using the directive #PRIO. The signal priorities will also effect the scheduling of processes, see "Scheduling" on page 3469.

Signal priorities can be specified, either:

- in the declaration of the signal
- in SDL output

It is impossible to specify #PRIO in a SDL input. Cmicro will ignore any occurrence of #PRIO in SDL input.

Signal priorities do affect the SDL output and the SDL create actions only.

The following rules are to be considered here:

- Signal priorities have to be in the range from 0 to 255.
- Signal priorities do not have to be consecutive, as process priorities have to be.
- If not specified otherwise (in SDL output) all instances of a signal have the same priority.
- Signal priority decreases with increasing numbers (zero is the highest priority level).
- The signal default priority is to be specified by the user (`xDefaultPrioSignal`), and must in the range of 0 to 255. As a recommendation, this value should be set to 100, so that both higher, as well as lower priorities can be declared with #PRIO.
- Signal priorities come after process priorities.
- If no #PRIO is specified for a signal, neither in its declaration, nor in any output, Cmicro uses `xDefaultPrioSignal` for each occurrence of that signal in an output.
- If #PRIO is specified only in the declaration of a signal, Cmicro uses this specified priority in each occurrence of that signal in an output.
- If #PRIO is specified in a specific output of a signal, but not in its declaration, then the specified #PRIO value is taken from the out-

put. If the signal is output without #PRIO, in that case `xDefaultPrioSignal` will be used.

- For dynamic process creation, an internal create signal is used. This signal carries the priority defined by `XMK_CREATE_PRIO`. As a general recommendation, this priority should be higher than any other signal priority.

The following example will give more explanations (note, that the values PA, PB are of sort pid):

Example 583: #PRIO Directive

```
Signal
S1, /*#PRIO 11 */
S2, /*#PRIO 22 */
S3,
S4; /*#PRIO 44 */

....
output S1 to PA
output S1 to PB; /*#PRIO 55*/
....
output S3 to PC;
output S3 to PD; /*#PRIO 66*/
```

Assuming the following C definition:

```
#define xDefaultPrioSignal 100
```

the following priorities will then be generated:

```
output S1 to PA--->use of prio 11
output S1 to PB--->use of prio 55
....
output S3 to PC--->use of prio 100
output S3 to PD--->use of prio 66
```

Modifying Outputs – Directive #EXTSIG, #ALT, #TRANSFER

There is no difference for the #EXTSIG, #ALT and #TRANSFER directives for Cmicro compared with Cadvanced, except that the use of it will sometime lead to a better performance. This is because if #EXTSIG for example is used in the case of an output to the environment, the user can prevent the Cmicro Kernel to be called (and the `xOutEnv` function to be executed).

Output of Code Generation

This section gives an overview of the code generated by the Cmicro SDL to C Compiler. This is useful, to make it possible to interpret the generated code. To know how the code is generated makes it quite easy to understand the program which is necessary and useful when testing and debugging erroneous executable programs.

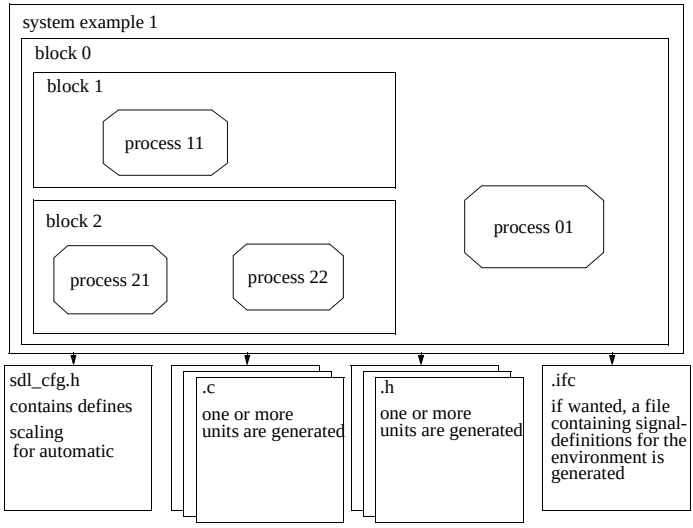


Figure 587: Structure of the generated C code

Not all the intricate details of the generated code are described here. The depth of description is sufficient to give the reader a reasonable understanding of the code generation algorithms. Explanations will illustrate what the code looks like, but not why.

The generated code contains several places where prefixes are generated, which consists of a prefix and unique numbering. The following prefix is generated for all objects: "`z<nnn>_`", where nnn is an incremental number.

Allowance for conditional compilation occurs in several places throughout the generated code. The generated C code is conditionally compiled, for example, for dynamic process creation (create symbol). A differentiation is made between conditional compilations generated by

the Cmicro SDL to C Compiler (called automatic scaling, prefix XMK_USED_) and conditional compilations which are dependent on header files, which are to be modified by the user (called manual scaling, prefix XMK_USE_).

> **Note:**
>
> Generally speaking, the ordering of the following subsections corresponds to the ordering in which the code is generated.

Each compilation unit is compiled either in one a.c file or into two files, a.c and a.h.

Only the differences are shown, when comparing the output of SDL to C Compiler with the Cmicro SDL to C Compiler. The overall differences of the both code generators are described in the section "Differences between Cmicro and Cadvanced" on page 3456 in chapter 66, *The Cmicro Library*.

Header of Generated C File

Code generation on the .c file for the current unit is started by generating the following header:

Example 584: The Head of a Generated C File

```
/* Program generated by the SDL Suite.Cmicro,
version x.y */
#define XSCT_CMICRO

#define C_MICRO_x_y
#define XMK_C_TIME <GenerationTime>
#include "ml_typ.h"
```

The XSCT_CMICRO macro can be used by the user to distinguish between the different Code generators, for example within ADT bodies.

The C_MICRO_x_y macro can be used by the user to distinguish between different versions of the Cmicro SDL to C Compiler. This is usually not but might become necessary if the output of the Cmicro SDL to C Compiler is different.

The XMK_C_TIME macro is used internally when compiling and linking and executing with the SDL Target Tester takes place. With this macro, a rough consistency check for the generated files is done. The

<GenerationTime> of the different files that are generated is compared in the Library and by the SDL Target Tester. If there is an inconsistency, compilation errors will occur.

The `#include "ml_typ.h"` is used to include all necessary declarations that the generated C code may use, including automatic scaling from `sdl_cfg.h` and predefined sorts.

SECTION Types and Forward References

As a difference to SDL to C compiler, this section contains the definitions for the process IDs and the forward declarations used in the generated C code.

Process IDs are generated as `#define` values in C, like:

```
#define XPTID_<UniquePrefix>_MyProcess 0
```

where the first process in the system is the value of 0 assigned, the second process gets the value 1, and so on. Please refer to "Generation of Identifiers" on page 3446 for more information.

The following forward references are generated:

```
extern XCONST XPDTBL yPDTBL_<UniquePrefix>_MyProcess;
```

Following this, the usual declarations are generated as described in *chapter 56, The Cadvanced/Cbasic SDL to C Compiler*.

No synonym variables are generated when using Cmicro.

Symbol Tables

Symbol tables are only generated for the SDL Target Tester, and not into the generated C code. The symbol tables generated for the SDL Target Tester are described within chapter 67, *The SDL Target Tester*.

Tables for Processes

Tables are used to represent the behavior of SDL objects, like processes and timers. It is not absolutely necessary to understand how these tables are generated and how the Cmicro Kernel works with them. The following subsections are only for those readers interested in the nature of the table structure.

Root Process Table

The root process table contains, for each of the defined SDL process types, a reference (i.e. a pointer) to the Process Description Table. The Cmicro Kernel is the main user of the root process table. Via this table, it can access all SDL process types and all SDL process instance data. The location of the generated root process table is directly before the yPAD-functions in the generated C file. The type definitions used in this table are located in the `ml_typ.h` module.

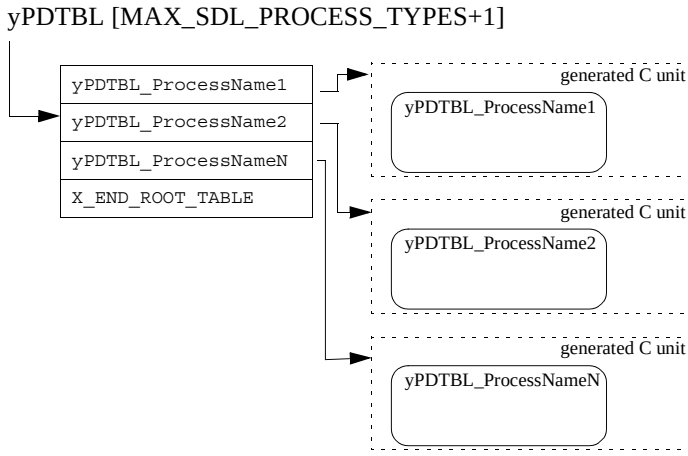


Figure 588: Root process table

Example 585: Code of Root Process Table ––––––––––––––––––––

C-Type definition (`ml_typ.h`):

```
extern xPDTBL yPDTBL [];/* for the Cmicro Kernel */
#define X_END_ROOT_TABLE/* Table-End Marker of yPDTBL*/
```

C constants (`sdl_cfg.h`):

```
#define MAX_SDL_PROCESS_TYPES <N>

   /* <Process-type-id´s>  Process Types are numbered */
   /* from 0 to N-1(see chapter "Generating PID") */
   #define XPTID_Process1Name 0
   #define XPTID_Process2Name 1
   #define XPTID_ProcessnName N-1
```

C code generation for the whole system:

```
XPDTBL yPDTBL [MAX_SDL_PROCESS_TYPES+1] =
{
  yPDTBL_ Process1Name,
  yPDTBL_ Process2Name,
  ......
  yPDTBL_ ProcessnName,
  X_END_ROOT_TABLE
}
```

Symbol Trace Table

In order to reduce the use of dynamic memory allocation, there is a table generated in the code which is used by the SDL Target Tester to store and retrieve test options, like switches, which define the trace.

The table is conditionally compiled and only included if the SDL Target Tester is contained in the target- executable.

The symbol trace table looks like:

Example 586: Code for Symbol Trace Table

```
/************************************************************
**  Symbol trace table
************************************************************/
#ifdef XMK_ADD_TEST_OPTIONS
XSYMTRACETBL *xSYMTRACETBL[MAX_SDL_PROCESS_TYPES+1] =
{
  (XSYMTRACETBL_ENTRY *)  NULL, /* for first Processtype */
  (XSYMTRACETBL_ENTRY *)  NULL, /* for second Processtype */
  ........
  ........
  (XSYMTRACETBL_ENTRY *)  NULL, /* for last Processtype */
  X_END_SYMTRACE_TABLE /* table end marker */
};
#endif
```

More information can be obtained by reading chapter 67, *The SDL Target Tester*.

Optimized Decision Trace information

An option to reduce the trace information for SDL decisions by showing only the first ten characters of the decision expression during trace. Setting the environment variable CMICRO_SHORT_DECISION_TRACE to any value prior to the Cmicro code generation is started, will have the effect on the generated C code that all xTraceDecision<parameter> statements will contain a parameter that is the first ten characters of the

decision expression instead of the complete expression. This will reduce the trace information for systems that contain a lot of decisions.

Instance-Data-Struct

The struct is generated in the header-section of the generated C file.

Example 587: Code Generation of type definition for each SDL process

```
typedef struct {
  PROCESS_VARS
  TypeName1  FPAR_var1;
  TypeName2  FPAR_var1;
  TypeName3  DCL_var1;
  TypeName4  DCL_var2;
  TypeName4  yExp_DCL_var2;
  TypeName5  FPAR_var1;
} yVDef_ProcessName;
```

Instances of a given type are represented as a C array. The code generation of variables for each SDL process looks like:

Example 588

```
#define X_MAX_INST_ProcessName upperlimitofprocessinstances1

static yVDef_ ProcessName

yINSTD_ProcessName[X_MAX_INST_ProcessName];
```

A reference to this array is generated in the Process Description Table which is discussed in the subsection "Process Description Table" on page 3427.

Process State Table

This table is generated for each process in the header-section of the generated C file. It contains information about the state of each process instance. The table contains ordinary SDL state values as well as the values `XSTARTUP` and `XDORMANT`. `XSTARTUP` is generated for each instance which is to be statically created (in (x, N) declarations, where x is > 0), `XDORMANT` is the value which is used to tag a process instance as sleeping. In the case of creation this instance can be reused.

Example 589: Code for Process State Table ----------

C typedef for the process state table (located in `ml_typ.h`):

```
typedef u_char xSTATE; /* see defines below */
#define XSTARTUP 0xff  /* valid only if xSTATE is */
                       /* u_char else 0xffff */
#define XDORMANT 0xfe  /* valid only if xSTATE is */
                       /* u_char, else 0xfffe */
```

C code generation for each process:

```
static xSTATE yPSTATETBL_znn_ProcessName
                                [X_MAX_INST_znn_ProcessName] =
{
  <creation-tag>  /* Instance 0 */
  <creation-tag>  /* Instance 1 */

  <creation-tag>  /* Instance M-1 */
};
```

where `<creation-tag>` is either `XSTARTUP` or `XDORMANT`.

Example 590: ----------

Code for a process type with 4 instances, 2 of which are to be created at SDL system start:

```
static xSTATE yPSTATETBL_znn_ProcessName [4] =
{
  XSTARTUP, /* Create at SDL-system-start */
  XSTARTUP, /* Create at SDL-system-start */
  XDORMANT, /* Create later               */
  XDORMANT  /* Create later               */
};
```

A reference to this table is created in the Process Description Table, which is discussed in the subsection "Process Description Table" on page 3427.

Transition Table

This is generated in the header-section of the generated C file. It contains all transitions of a process, including asterisk states, asterisk inputs and asterisk save.

The C `typedef` for the transition table (located in `ml_typ.h`) is as follows:

Example 591: Code for Transition Table ––––––––––––––––––––––

```
typedef struct {
  xINPUT    SignalID; /* Input, Asterisk-Input. */
                      /* Input is Timer */

                      /* and/or ordinary Signal */
  xSYMBOLNR SymbolNr; /* Symbolnumber to be used */
                      /* in yPAD-function */
} xTR_TABLE_ENTRY;
```

C code generation:

```
static XCONST xTR_TABLE_ENTRY   yTRTBL_znn_ProcessName
                                [XMAX_TRANS_znn_ProcessName]=
{
  /* state_0-table */
  input_1, SymbolNr,
  input_2, SymbolNr,
  XASTERISK,XSAVEID /* asterisk save */

  input_N, SymbolNr,

  /* state_1-table */
  ..............
  ..............
  /* state_j-table */
  input_1, SymbolNr,
  input_2, SymbolNr,

  input_N, trans_jN,
  XASTERISK,XSAVEID /* asterisk save */
};
```

The `SymbolNr` shown above is used to select the right transition in the switch generated in the yPAD function.

Where the C define

`XASTERISK` is an ID defining all possible SDL Inputs (asterisk Inputs),

`XSAVEID` is a simple ID defined in `ml_typ.h` which can be compared by the SDL Kernel to detect signal-save.

And where:

```
#define XASTERISK   -1
#define XSAVEID     xSave
```

––

A reference to this table is created in the Process Description Table.

State Index Table

This is generated in the header section of the generated C file.

Example 592: Code for State Index Table

C typedef (`ml_typ.h`):

```
typedef   u_char xSTATE_INDEX;
```

C code generation (header of generated C file):

```
static xCONST xSITBL   xSTATE_INDEX_znn_ProcessName
                         [<count_transitions_of_ProcessName] =
{
  0, /* i.e.a process with 3 states, but no asterisk states */
     /* state_0 has 2 transitions */
  2, /* state_1 has 5 transitions */
  7, /* state_2 has 3 transitions */
  10 /* table-end-index XI_TABLE_END */
};
```

The first value in the above table indicates the beginning of the first state in the Transition Table. If asterisk state definitions are not found in the process, this value is 0.

A reference to this table is created in the Process Description Table.

PID Table

These tables are used to store the values parent and offspring for each process. The reason an extra table is used to store this information is to simplify initialization. The Cmicro Kernel updates the values in the table according to the SDL rules.

Example 593: Code for PID Table

C-type definition (`ml_typ.h`):

```
#ifdef XMK_USE_PID_ADDRESSING
  typedef struct
  {
    #ifdef XMK_USE_SDL_PARENT
      xPID            Parent;
    #endif

    #ifdef XMK_USE_SDL_OFFSPRING
      xPID            Offspring;
    #endif

  } xPIDTable;
#endif
```

C code generation for each process:

```
/*-----------Process-PID-Values-------------*/
#ifdef XMK_USE_PID_ADDRESSING
  static xPIDTable yPID_TBL_z00_P1[X_MAX_INST_z00_P1];
#endif
```

A reference to this table is created in the Process Description Table, which is discussed in the subsection "Process Description Table" on page 3427.

Process Description Table

For each SDL process, an automatically initialized C structure is generated called process description table. This table is used in the Root Process Table to enable the Cmicro Kernel to access process type information as well as process instance data.

Inspect the following diagram to see which information is contained in the process description table:

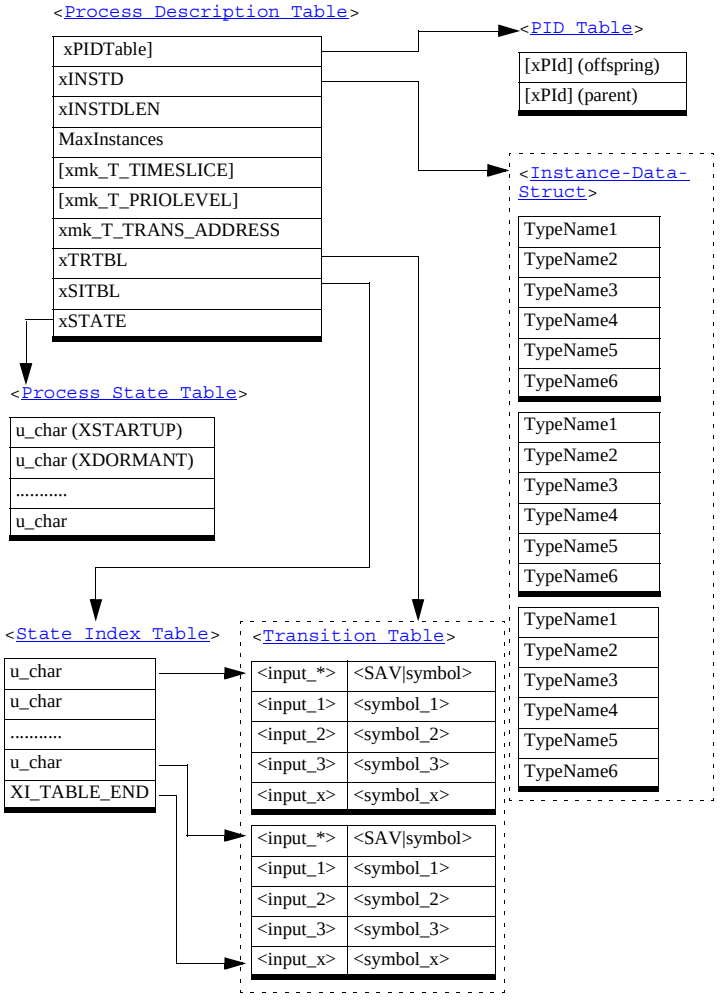


Figure 589: Process description table

Allocated to each SDL process type is one table `yPDTBL_ProcessName`.

The type definitions of this table are located in the `ml_typ.h` module.

Example 594: Code for Process Description Table

C typedef for the process description table (`ml_typ.h`):

```
typedef struct {
  #ifdef XMK_USE_PID_ADDRESSING
    xPIDTable  *pPIDTable; /* Table with */
                           /* Parent/OffspringValues */
  #endif

  xINSTD      *pInstanceData ; /* Pointer to Instancedata*/
                               /* Vector                 */
  xINSTDLEN    DataLength ;  /* Length of Instancedata  */
                             /* for 1 Instance          */
                             /* (used by SDL-BS)        */
  unsigned char MaxInstances ; /* Max.Number of Instances*/

  #ifdef XMK_USE_TIMESLICE
    /* Time-Slices can be individually specified by the user*/
    /* The value stored in TimeSlice is measured in ticks   */
    /* The Cmicro Kernel has to be scaled to handle     */
    /* timeslicing */
    xmk_T_TIMESLICE TimeSlice;
  #endif

  #ifdef XMK_USE_PREEMPTIVE
    /* Process-Priority can be specified with #PRIO on the */
    /* SDL-Level. It is available only, if the Cmicro    */
    /* Kernel is scaled to handle preemption. */
    xmk_T_PRIOLEVEL PrioLevel;/*Priority of this processtype*/
  #endif

  xmk_T_TRANS_ADDRESS yPAD_Function ; /* Address of the */
                                      /* yPADFunction */
  xTRTBL TransitionTable ;  /* Pointer to transition table */
  xSITBL *StateIndexTable ; /* Pointer to state index table */
  xSTATE *ProcessStateTable;/* Pointer to process state table
*/
} XPDTBL;
```

C code generation for each process:

```
#define X_MAX_INST_ProcessName 1

xPDTBL yPDTBL_ProcessName =
{
  yPID_TBL_znn_<process:N>,
  (xINSTD*) yINSTD_znn_ProcessName,
  X_MAX_INST_znn_ProcessName,
  (xmk_T_TRANS_ADDRESS) yPAD_znn_ProcessName,
  yTRTBL_znn_ProcessName;
  xSTATE_INDEX_znn_ProcessName,
  yPSTATETBL_znn_ProcessName;
};
```

For each generated process description table, a new entry in the Root Process Table is generated.

Actions by Processes and Procedures

GR References

No code is generated to evaluate the graphical references during runtime of the SDL system. A large amount of memory is required to store and handle such information which normally proves too large for any real target system.

Alternatively, C comments are generated which make it possible to verify and debug the generated code as illustrated in the following example. The PR <position> indicates in which line number of the SDL/PR file the symbol can be found.

```
For processes :
/*************************************************************
**  PROCESS <process-name>
**  <<SYSTEM <system-name>/BLOCK <block-name>>
**  #SDTREF(<reference>)
*************************************************************/

For signals :
/*************************************************************
**  SIGNAL S1
**  <<SYSTEM <system-name>/BLOCK <block-name>>
**  #SDTREF(<reference>)
*************************************************************/

For yPAD-function
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
** Function for process <process-name>
** #SDTREF(<reference>)
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

For output :
/*-----
** OUTPUT <signal-name>
** #SDTREF(<reference>)
------*/

For nextstate :
/*-----
** NEXTSTATE <state-name>
** #SDTREF(<reference>)
------*/
```

Structure of Process and Procedure Functions

The basic structure of the generated C code for process and procedure definitions remains the same as for the SDL to C compiler although some modifications are evident.

The code generation for the PAD function is different compared with Cadvanced, in the way that code that is common in process types is copied into the PAD function for instantiated processes.

Procedures follow the same code generation as processes, with some small exceptions in macro naming conventions for variable declarations.

Each SDL process is represented in C by a C function called `yPAD_ProcessName`.

Example 595: `yPAD_ProcessName` ----------

```
/* Function for process ProcessName */
#ifndef XNOPROTO
extern YPAD_RESULT_TYPE yPAD_ProcessName ( YPAD_ANSI_PARAM )
#else
extern YPAD_RESULT_TYPE yPAD_ProcessName ( YPAD_KR_PARAM )
  YPAD_KR_DEF
#endif
{
  local variable section
  State-input-selection
  {
    start-transition including nextstate
    transition-1 including nextstate
    transition-2 including nextstate
    .......
    transition-n including nextstate
  }
  pad-end-section
}

/* Function for procedure ProcedureName */
#ifndef XNOPROTO
extern YPRD_RESULT_TYPE yPAD_ProcedureName ( YPRD_ANSI_PARAM )
#else
extern YPRD_RESULT_TYPE yPAD_ProcedureName ( YPRD_KR_PARAM )
  YPRD_KR_DEF
#endif
{
  local variable section
  section representing procedure body
}
```

Local Variables Section

The following defines are generated in the local variables section for processes.

Example 596 ----------

```
YPAD_YSVARP             /* used for signal variable pointers
*/
YPAD_YVARP(yVDef_z00_P1)/* used for process variables */
YPAD_TEMP_VARS          /* used for temporary variables */
YPRSNAME_VAR("P1")      /* can be used for printf */
BEGIN_PAD               /* used for some preparations */
                        /* to handle signals, or Integration*/
                        /* of any Realtime operating system */
```

After expansion by the C preprocessor:

```
yVDef_z00_ProcessName *yVarP
                          =(yVDef_z00_ProcessName *)pRunData;
unsigned char *yOutputSignal;
unsigned char *ySVarP;

(void) printf(("PROCESS:%s\n", "ProcessName"));

if((P_MESSAGE != ((void *) 0))
    && (P_MESSAGE->mess_length > 4))
{
  ySVarP = (unsigned char *) P_MESSAGE->mess_ud.pt_ud;
}
else
{
  ySVarP = (unsigned char *) P_MESSAGE->mess_ud.ud;
}
```

The following defines are generated in the local variables section for procedures:

```
YPRD_YVARP(yVDef_znnn_ProcedureName)
/* used for procedure variables */

YPRD_TEMP_VARS
/* used for temporary variables */

YPRDNAME_VAR("ProcedureName")
/* can be used for printf */
```

State – Input Selection

The selection of the appropriate SDL transition which is to be executed in the current state with the current signal in the input port goes in principle over the transition table, described in previous chapters. With this table, the Cmicro Kernel can evaluate a symbol number, which is local to a process, a unique numbering of the different possible transitions. This numbering algorithm begins at 0 (which corresponds to the start symbol) and continues until all symbols for this particular process type have been numbered.

The appropriate transition is selected by the following switch:

```
switch (XSYMBOLNUMBER) {
{
  case 0:. start-transition
  nextstate;

  case 1: transition-1
  nextstate;
}
```

After pre-compiling it:

```
switch (_xSymbolNumber_ )
{
  .....
}
```

Start Transition

The start transition is included into the body of the generated yPAD function and has the same layout as transitions, with the following exceptions:

Assignment of initialization values to all local variables in the processes and procedures (if any) is executed. All DCL variables are filled with their default-values.

The start transition is selected by the special case-value zero in the switch-statement of the yPAD function.

> **Note:**
>
> FPARS in dynamic process creation are not contained in this version of the Cmicro Package.

Transitions

The transitions are translated in the order they are found and are only translated to the sequence of actions they consist of. The translation of actions are discussed in the subsection "Translation of Actions" on page 3434 following a few lines below.

PAD-End-Section

Each yPAD function is finished with:

```
END_PAD (yPAD_ProcessName);
```

The main reason for this is to make it possible to integrate other real-time operating systems.

> **Note:**
>
> Some compilers produce a warning if there is no return at the end of the yPAD function. Other compilers produce a warning "unreachable code", if there is a return at the end of the yPAD function. For this reason, a function returning macro `END_PAD` exists which can be expanded in accordance with the particular compiler used.

Translation of Actions

Translation of Output

SDL output statements are translated to the following basic structure:

- allocate the data area for the parameters of the signal to be output
- assign signal parameters
- send the signal, parameters will be copied
- release the data area for the parameters of the signal.

There are a lot of different output macros generated. The main reason for this is that for each output situation an optimized code is to be generated.

One differentiation is made for signals without parameters and signals with parameters. For a signal without parameters, suffix `_NPAR` is used for the macro generated and for a signal with parameters, suffix `_PAR` is used. The relevant output macro can then be expanded to a simpler output C function called `xmk_SendSimple`, if no signal priority is used.

Another differentiation is made for signals which are sent to the system's environment or which are sent internally in the SDL system. The suffix `_ENV` is appended to the macros which are shown here, if the signal should go to the system environment.

The different directives which can be used within the SDL Suite to modify outputs are discussed in subsection "Modifying Outputs – Directive #EXTSIG, #ALT, #TRANSFER" on page 3417.

The other different output situations which are handled, will be described in the next subsections.

Output without TO and without VIA

If the user specifies output SignalName without TO and VIA in SDL, the Cmicro SDL to C Compiler calculates the receiver of the signal. It is also possible to have more than one receiver for the signal. During execution time, any possible receiver that are alive may be selected otherwise if no receiver can be found, the C function `ErrorHandler` will be called. The following code is generated:

```
ALLOC_SIGNAL_ppp(SignalNamewithoutPrefix,
                 SignalNamewithPrefix,
                 SignalParameterTypeStructureName)
```

ordinary assignment of Signal Parameters, if there are some...

```
SDL_OUTP_ppp(Priority,
             SignalNamewithoutPrefix,
             SignalNamewithPrefix,
             TO_PROCESS(ProcessNamewithoutPrefix,
                        ProcessNamewithPrefix),
             SignalParameterTypeStructureName,
             "SignalNamewithoutPrefix")
```

Note:

The `ppp` above stands for either `PAR` or `NPAR` for a Signal with or without parameters.

After expansion, the user will find a C function call to the `xmk_SendSimple` function or the `xmk_Send` function.

Priority is generated as `xDefaultPrioSignal` if no priority is specified for the signal with `#PRIO`.

`TO_PROCESS` is expanded to a function call if there is at minimum one (x, N) declaration in the system, where N is > 1. This function returns one of the possible receivers of the signal.

`TO_PROCESS` selects an active instance of the given process type It does not check for different types as receivers.

`TO_PROCESS` is expanded so that the pid is passed directly to one of the C functions `xmk_Send*`, if there are only (x,1) declarations in the system.

If the environment is the receiver of the signal, then the following code is generated:

```
ALLOC_SIGNAL_ppp(SignalNamewithoutPrefix,
                 SignalNamewithPrefix,
                 SignalParameterTypeStructureName)
```

ordinary assignment of Signal Parameters, if there are any...

```
SDL_OUTP_ppp_ENV(Priority,
                 SignalNamewithoutPrefix,
                 SignalNamewithPrefix,
                 ENV,
                 SignalParameterTypeStructureName,
       "SignalNamewithoutPrefix")
```

Note:

The `ppp` above stands for either `PAR` or `NPAR` for a Signal with or without parameters.

After expansion, the user will find that `ENV` is passed to one of the C functions `xmk_SendSimple` or `xmk_Send`. `ENV` is a special value used

inside the Cmicro Kernel to detect which signals are to be passed to the C function `xOutEnv`.

Output with TO clause

If the user specifies the output SignalName to pid in SDL, the Cmicro SDL to C Compiler generates the following code:

```
ALLOC_SIGNAL_ppp(SignalNamewithoutPrefix,
                 SignalNamewithPrefix,
                 SignalParameterTypeStructureName)
```

ordinary assignment of Signal Parameters, if there are some...

```
SDL_OUTP_ppp(Priority,
             SignalNamewithoutPrefix,
             SignalNamewithPrefix,
             pid-variable,
             SignalParameterTypeStructureName,
             "SignalNamewithoutPrefix")
```

Note:

The `ppp` above either stands for `PAR` or `NPAR` for a Signal with or without parameters.

Expansion reveals a C function call to the `xmk_SendSimple` function or the `xmk_Send` function.

Priority is generated as `xDefaultPrioSignal`, if no priority is specified for the signal with #PRIO.

Possible generated values for pid variable are `SDL_SENDER`, `SDL_PARENT`, `SDL_OFFSPRING` and `SDL_SELF` or an SDL pid variable. These values are passed to the `xmk_Send*` functions. The name of a process as specified in SDL may also be given.

Output with VIA clause

The Cmicro SDL to C Compiler computes the possible receivers in an output with the **VIA** clause. If there are several possible receivers, an error message is produced.

If there is exactly one receiver, the same code is generated as for SDL output without **to**.

List of Generated Output Macros

- `ALLOC_SIGNAL_NPAR`
  Allocating memory for signal without parameters
- `ALLOC_SIGNAL_PAR`
  same for signals with parameters
- `TO_PROCESS`
  Macro used to evaluate a receiver process instance, if necessary in the case of (x, N) declarations, where N > 1.
- `SDL_OUTP_NPAR`
  Output internally in the SDL system for signal without parameters
- `SDL_OUTP_PAR`
  same for signal with parameters
- `SDL_OUTP_NPAR_ENV`
  Output to the system environment for signal without parameters
- `SDL_OUTP_PAR_ENV`
  same for signal with parameters
- `SDL_ALTOUTP_NPAR`
  #ALT for an output internally in the SDL system for signal without parameters
- `SDL_ALTOUTP_PAR`
  same for signal with parameters
- `SDL_ALTOUTP_NPAR_ENV`
  #ALT for an output to the system environment for signal without parameters
- `SDL_ALTOUTP_PAR_ENV`
  same for signal with parameters
- `EXT_SignalName`
  if #EXTSIG is used in output
- `TRANSFER_SIGNAL`
  #TRANSFER is used in output

Translation of Create

The create action in SDL is translated to the following C code:

```
ALLOC_STARTUP_ppp(ProcessNamewithoutPrefix,
                  ProcessNamewithPrefix,
                  "ProcessNamewithoutPrefix, 0);
```

....assignment of start-up values (cannot be used in this version of the Cmicro Package)

```
SDL_CREATE(ProcessNamewithoutPrefix,
           ProcessNamewithPrefix,
           "ProcessNamewithoutPrefix, 0,
           VariableofCreatedProcess,
           PriorityofCreatedProcess,
           yPAD-functionNameofCreatedProcess);
```

`PriorityofCreatedProcess` is generated as `xDefaultPrioProcess`, if no priority is specified with #PRIO.

Translation of Set

The translation of set is restricted in a few areas in order to produce efficient code for a micro controller. For example, the SDL duration expressed by a real value in the context of timers is not implemented. The reason for this is that controllers do not have floating point operations or floating point operations are not used in order to increase the performance. For timers, such a high resolution is not necessary in most applications. The Cmicro Package uses a long value in its standard implementation to represent absolute time.

In order to make the examples below more readable, it is assumed that at least one timer with parameter is used in the system (macro `XMK_USED_TIMER_WITH_PARAMS` is defined in the generated file `sdl_cfg.h`). If the macro is not defined, then the handling for timers with parameters is not included.

Example 597 ----------

If the following is specified in SDL/PR:

```
Timer TimerName;
        ......
Set (now + durationvalue, TimerName) ;
```

or

```
Set (now + 22222, TimerName) ;
```

then the following code is generated:

```
SDL_SET_DUR  \
(xPlus_SDL_Time(SDL_NOW,SDL_DURATION_LIT(22222.0,22222,0)),
           SDL_DURATION_LIT(22222.0, 22222, 0),
           TimerName,
           TimerNamewithPrefix,
           yTim_timer2,
           "TimerNamewithoutPrefix")
```

Example 598 ----------

If the following is specified in SDL/PR:

```
Timer TimerName := TimerGroundValue ;---> see Note: on page 3440!
```

then the following code is generated:

```
SDL_SET_TICKS
(xPlus_SDL_Time(SDL_NOW, TICKS(SDL_INTEGER_LIT(22222))),
              TICKS(SDL_INTEGER_LIT(22222)),
              TimerName,
              TimerNamewithPrefix,
              yTim_timer2,
              "TimerNamewithoutPrefix")
```

The code after expansion then contains a function call to

```
xmk_TimerSet (TIMEEXPR,TimerNamewithPrefix,0).
```

`TIMEEXPR` is the result of the evaluation of **now** plus duration value.

```
#define SDL_SET_DUR(TIME_EXPR, DUR_EXPR, TIMER_NAME,
        TIMER_IDNODE, TIMER_VAR, TIMER_NAME_STRING) \
xmk_TimerSet(TIME_EXPR, TIMER_IDNODE,0);

#define SDL_SET_TICKS(TIME_EXPR, DUR_EXPR, TIMER_NAME,
        TIMER_IDNODE, TIMER_VAR, TIMER_NAME_STRING) \
xmk_TimerSet(TIME_EXPR,TIMER_IDNODE,0);
```

Example 599 ----------

If a timer with parameter is defined in SDL/PR:

```
Timer TimerName (integer);
...
set (now+1, TimerName (4711));
```

then the following code is generated:

```
SDL_SET_DUR_WITH_1IPARA(xPlus_SDL_Time(SDL_NOW,
                       SDL_DURATION_LIT(1.0, 1, 0)),
                       SDL_DURATION_LIT(1.0, 1, 0), TimerName,
                       TimerNamewithPrefix,
                       yPDef_z262_twp1,
                       yTim_TimerName,
                       "TimerName",
                       SDL_INTEGER_LIT(4711))
```

The code after expansion then contains a function call to

```
xmk_TimerSet (TIMEEXPR,TimerNamewithPrefix,4711).
```

Restrictions in the Use of Timers

- Timers with parameters are restrictively supported in Cmicro. There might be only one parameter of sort “integer”. This implementation has been chosen to achieve the highest efficiency.
- Duration values as real values are not supported in this version of the Cmicro Package, i.e. this:

```
set (now + 5.5, TimerName)
```

  is **not allowed** (the real part is discarded i.e. 5.5 (= 5).

Translation of Reset

If the user specifies in SDL/PR:

```
Reset (TimerName) ;
```

then the following code is generated:

```
SDL_RESET(TimerNamewithoutPrefix,
          TimerNamewithPrefix,
          yTim_TimerName)
```

The code after expansion contains a function call to `xmk_TimerReset` (TimerNamewithPrefix).

For a timer with one integer parameter, the following macro call is generated:

```
SDL_RESET_WITH_1IPARA(TimerNamewithoutPrefix,
                      TimerNamewithPrefix,
                      TimerParStruct,
                      yTim_TimerName,
                      TimerValue)
```

Note:

Timers with parameters are supported with the restriction that only one integer parameter is allowed.

Translation of Call

As SDL procedures are implemented with the restrictions explained within subsection “SDL Restrictions” on page 3450, the following explanatory C code (to a procedure called `ex_proc`) is generated:

```
ex_proc (....C parameters ...);
```

All necessary parameters are routed via the C function call stack.

Translation of Call to a Procedure Returning Value / Operator Diagram

Operator diagrams and procedures returning values are – considering the call – handled in the same way please see the following explanatory example:

Example 600: Procedure Call ––––––––––––––––––––––––––––––

```
TASK i := (call p(1)) + (call Q(i,k));
```

is translated to something like:

```
i = p(1) + Q(i,k);
```

––

> **Note:**
> The value of returning procedure calls are transformed to C functions returning values.

Translation of Nextstate

The nextstate operation is generated at the end of each transition contained in the yPAD function, as follows:

- If the process performs simple nextstate operation:

  ```
  SDL_NEXTSTATE(State1, z000_State1, "State1")
  ```

  after preprocessing:

  ```
  return (z000_State1);
  ```

- If it performs a nextstate, which is defined as a dash state:

  ```
  SDL_DASH_NEXTSTATE
  ```

  which is defined as:

  ```
  return (XDASHSTATE);
  ```

Translation of Stop

A stop action is translated to:

```
SDL_STOP
```

which is defined as

```
return (XDORMANT);
```

which is good code saving. The Cmicro Kernel then enters the new state value into the Process State Table.

> **Note:**
>
> This table contains ordinary SDL state values as well as the values `XSTARTUP` and `XDORMANT`. `XSTARTUP` is generated for each instance which is to be statically created (in (x, N) declarations, where x is > 0). `XDORMANT` is the value which is used to tag a process instance as sleeping. In the case of creation this instance can be reused.

Translation of Return

```
#ifdef XFREEVARS
  FREE_PROCESS_VARS ()
#endif

SDL_RETURN
```

The macro definitions are:

```
#define SDL_RETURN \
        if (_xxptr != (unsigned char*) NULL) \
        { \
          XMK_MEM_FREE ((unsigned char *)_xxptr); \
        } \
        return ;
```

where xxptr is the pointer to the procedure instance data, as given via the C function call parameter list. Note, that the memory previously allocated directly before the procedure call is freed at the end of the procedure, not outside of the procedure.

Translation of SDL Expressions

In this section some of the translation rules for expressions are described. For more information see *"Translation of Sorts" on page 2665 in chapter 56, The Cadvanced/Cbasic SDL to C Compiler* where for example the translation rules for literals and operators in the predefined abstract data types are given.

Now

SDL now is translated to the macro `SDL_NOW` which is expanded to the C function `xmk_NOW`. This function is exported by the module `mk_stim.c`.

Self, Parent, Offspring, Sender

The definitions for **self**, **parent**, **offspring**, **sender** are:

```
#ifdef  XMK_USED_SELF
#define SDL_SELF               xRunPID
#endif

#ifdef  XMK_USED_PARENT
#define SDL_PARENT             pRunPIDTable->Parent
#endif

#ifdef  XMK_USED_OFFSPRING
#define SDL_OFFSPRING          pRunPIDTable->Offspring
#endif

#ifdef  XMK_USED_SENDER
#define SDL_SENDER             P_MESSAGE->send
#endif
```

All the variables above are of type `xPID`. All variables are maintained by the Cmicro Kernel. `xRunPID` is a global variable which contains the pid of the SDL process which is currently running. `P_MESSAGE` is a pointer to the signal instance which is currently worked on.

Timer Active

An SDL timer active expression is translated to:

```
SDL_ACTIVE(TimerName, TimerName,
  yTim_TimerName)
```

which is expanded to:

```
xmk_TimerActive(TimerName)
```

A conditional expression in SDL is translated to a conditional expression in C.

Init Function

An explicit initialization function is not generated by the Cmicro SDL to C Compiler in any case.

The structure of the SDL system is not generated into the C code. What is seen in the generated code, is the behavior of the SDL system. Vari-

ables of processes are initialized during the start transition of a process and no information about the structure of the SDL system is available during run-time in the generated code.

An initialization function is generated only in that case if synonyms are used within SDL, which require an initialized C variable.

All this results in a more compact executable.

For example, the following use of an SDL synonym results in a generated initialization function:

```
synonym a integer := /*#CODE anyUserFunction () */
```

The following C code is then generated within the C function `yInit`:

```
yAssF_SDL_Integer(a, anyUserFunction (), XASS);
```

`yInit` is called by the Cmicro Kernel if the define

```
XMK_USED_INITFUNC
```

is generated into the file `sdl_cfg.h`, which is done in the case above.

Initialization of Synonyms

The Cmicro SDL to C Compiler allows SDL synonyms to be implemented as C macros and C variables.

Initialization is implemented within the C function `yInit` which is conditionally compiled.

Function main

The C function `main` is not automatically generated by the Cmicro SDL to C Compiler. This is unnecessary because the `main` function usually is provided from the user or the predefined `main` function can be used. Instead of an automatically generated `main` function, the user must supply the function body of `main`, for target applications. Guidelines can be found in the subsection "Implementation of Main Function" on page 3530 in chapter 66, *The Cmicro Library*.

Symbol Table File

The structure of an SDL system can be represented by a tree diagram. In SDL the root of the tree is represented by the SDL system followed by blocks, block substructures, processes and procedures[1]. Channels, channel substructures and signal routes are also represented in the tree. This tree is static, which means it cannot be modified during the run-time of an SDL system.

The SDL to C Compiler generates code so that this static structure is present in the generated code. This is good for debugging purposes.

The Cmicro SDL to C Compiler generates code so that this static structure is **not** present in the generated code, in order to spare memory. To enable debugging of the generated code, C comments are generated. Please consult the subsection "GR References" on page 3430.

A symbol table is necessary for the SDL Target Tester running on the host or the development system.

For more information consult chapter 67, *The SDL Target Tester*.

1. In SDL-92 several SDL systems can exist in parallel.

Generation of Identifiers

Processes and Process IDs (PID)

In order to implement the environment functions, it is important to notice that process IDs in Cmicro are generated into the `sdl_cfg.h` file, which must be included by the user's environment C module. These IDs are coded like it is described in "Generated Configuration File" on page 3394. More explanations are given in the following.

Since process IDs might become ambiguous, especially in block type and process type instantiations in SDL'92, the names of process IDs that are to be used in the environment functions are to be given a prefix. Using this prefix within the environment functions (`xInEnv`), it can be guaranteed that different process IDs (equates to "instance sets" in SDL'92) with the same name can be distinguished, which is necessary in order to send signals to the right process instance within the SDL system. On the other hand, prefixes are not necessary when all the process instance sets within the system have a different name. The Cmicro SDL to C Compiler uses an algorithm to calculate the prefixes in the most convenient way.

For example, if a process named "myprocess" exists only once within the SDL system, there will be no automatic prefix generated, e.g. the full process ID is

```
#define XPTID_myprocess 0
```

If, as another example, the process "myprocess" exists twice, for example once within a block called "myfirstblock" and once more within a block called "mysecondblock", the Cmicro SDL to C Compiler then creates two definitions which guarantee that the processes can be distinguished:

```
#define XPTID_myfirstblock_myprocess  0
#define XPTID_mysecondblock_myprocess 1
```

In this way, by adding scope names (block names), prefixes are always generated in a way so that no naming conflicts occur. Of course, for process and block type instantiations, the name of the instance is being used to generate this unambiguous prefix.

SDL process types (process instance sets in SDL'92), as well as SDL process instances are numbered consecutively beginning with zero. The

ordering of these numbers is the same as the ordering of the processes in the SDL/PR file.

The values 250 to 255 are reserved for internal purposes and must not be used for process type numbering. The Targeting Expert checks this rule automatically. For small systems this does not create any problems.

The Cmicro Kernel assumes the above definitions.

In the generated C code, the SDL values self, sender, parent and offspring, and variables of this type are represented by the `typedef xPID`. The intention is to have unique numbering of processes and their instances in the whole SDL system. This becomes necessary because of the Cmicro Code no longer containing the structure of the SDL system (system, block...). The `typedef xPID` is defined as

- `unsigned char` or `unsigned int`
  if there are only (x,1) declarations in the system no distinction between instances is necessary. This is automatically detected. See the flag `XMK_USED_ONLY_X_1` in the section "Automatic Scaling Included in Cmicro" on page 3519 in chapter 66, *The Cmicro Library*.

- `unsigned int` or `unsigned long`
  if there is at minimum one (x, N) declaration in the system, where N > 1, instances need to be distinguishable from each other.

There are a few macros defined to extract the process type number or the process instance number from a variable of the type `xPID` and to build an `xPID` variable from a process type number and a process instance number, the users do not have to think about the internal representation:

Example 601: Macros to extract process type or instance number ––

```
processtype     = EPIDTYPE(xPID_variable)
processinstance = EPIDINST(xPID_variable)
xPID_variable   = GLOBALPID(processtype, processinstance)
```

––

Signals and Timers

SDL signals and timers are numbered automatically by the Cmicro SDL to C Compiler so that they have a unique number over the complete system. Timers are represented by the values 1, 2, 3.... MT to the last timer of the MT timers in the system. After that follow ordinary SDL signal numbers beginning with MT+1, MT+2, MT+3... MT + MS.

When using the standard Cmicro Package, as delivered, then the values 0 and 251 to 255 are reserved for internal purposes. If the upper limit of 250 signals and timers is being reached, then the signal ID type has to be changed from `unsigned char` to `unsigned int`, thus allowing more than 60000 signals/timers to be handled. All these changes will be done if the flag XMK_USE_MORE_THAN_250_SIGNALS is set.

Caution!

The Cmicro SDL to C Compiler does not check for the upper limit of 250 signals being reached for a generated SDL system. Instead the Targeting Expert will check the amount of signals and timers in the SDL system and will inform the user.

Example 602: ----------

C code generated for signals and timers:

```
#define znnn_SignalName 1
#define znnn_SignalName 2
```

Where `znnn_` is the automatically generated prefix which is required to cope with the SDL scope rules. Remember, that processes in SDL can have the same name as signals, states etc. Prefixing, however, ensures uniquely named SDL objects in the generated C Code.

Example 603: ----------

A system with 2 signals S1 and S2, and a timer TIMER1:

```
#define z049_TIMER1     1
#define z050_S1         2
#define z051_S2         3
```

When it comes to connecting the environment to the SDL system, the automatic numbering of signal IDs and timer IDs may not be required. If the user wants to prevent the automatic numbering of signals, then it is possible to #include a file containing all the signal and timer numbers. The file may contain something like:

```
#undef  SignalOrTimerName
#define SignalOrTimerName AnyValueAccordingToKernelRules
```

States

SDL states are consecutively numbered from 1 through to N for each process type. The values 0, and 250 to 255 are reserved for internal purposes in the Cmicro Package. This restriction incurs no foreseeable difficulty as processes should never have more than 50 States as a recommendation.

If there are even more states per process the flag XMK_USE_HUGE_TRANSITIONTABLES must be set.

The following C code generation is supplied for the header-section of the generated C file(s).

For each SDL process:

```
#define znnn_State1Name 1
#define znnn_State2Name 2
....
#define znnn_State3Name 3
```

Example 604: ----------

For a process with 2 states S1 and S2:

```
#define z020_S1
#define z021_S2
```

These values are used in the state-index-table and in the generated C functions, wherever a nextstate is referenced.

SDL Restrictions

General

The Cmicro SDL to C Compiler handles SDL concepts according to the definition of SDL-92. In addition to the restrictions of all the SDL to C Compilers, the following additional restrictions are introduced for the Cmicro SDL to C Compiler:

- Inheritance of procedures
- Procedures with states
- Remote Procedure Calls
- Nested procedure call data scope
- Export / Import
- View / Reveal
- Enabling condition / Continuous signal
- Service and priority input and output
- Channel substructure
- Declaring an infinite number of process instances (x,) or (,)
- FPARS when creating a process
- Omission of parameters in a signal input
- Output via all
- Timers duration values cannot be real
- Timers with more than one parameter
- Timers with another parameter than sort integer
- The any expression
- Only the list of ADT and packages that are explained in the subsection "Exceptions for SDL Predefined Types" on page 3406 and the subsection "Exceptions for Implementations of Operators" on page 3409 are handled correctly with Cmicro.

The following restrictions are additional regarding the packages that are delivered together with the SDL Suite.

sdth2sdl

It is impossible to read in header files created with Cadvanced and use them in Cmicro and the other way around. The reason is that it is impossible to mix up C code between Cadvanced/Cbasic and Cmicro.

Combining Cadvanced / Cmicro C Code

Mixing C code from different C Code Generators is not possible as the different code generators use their own run-time model and run-time data structures. Trying to mix up the C code will lead to compilation errors.

Light and Tight Target Integrations

The light and tight target integrations delivered with the SDL Suite are only available for Cadvanced but not applicable to Cmicro. There are light and tight target integrations for Cmicro but these are not part of the product.

Restrictions in Combination with SDL Target Tester

Scope Rules / Qualifiers

If the SDL Target Tester is to be used, then the **scope rules** of SDL are handled in a restrictive fashion. No information is generated for the system, block, block substructure, channel and signalroute. After applying the Cmicro SDL to C Compiler, all the structuring information is lost.

This means that it is impossible to address two different processes with the same name in different blocks. In order to avoid problems, give all processes, signals and timers in the system a different (unique) name.

Predefined Sorts

The predefined sort charstring and all the predefined sort that are based on the implementation of charstring (like predefined sorts from ASN.1) cannot be handled, if the SDL Target Tester is to be used. All predefined sorts which are generated into pointers in C cannot be used. In order to get a detailed description, please see in chapter 67, *The SDL Target Tester*.

Analyzer Restrictions

The restrictions in the SDL Analyzer, which also affects the Cmicro SDL to C Compiler, are summarized in *chapter 54, The SDL Analyzer*.

Declaration of signals

There is a restriction in the Cmicro code generator which causes a compilation error. The error is related to declaration of signals under the three following conditions:

- The signal must be declared inside a block. If the signal is declared on system or package level, this restriction will not occur.
- The signal must have parameters that are deallocated by the receiving process. For signals containing simple types or no parameters, this restriction will not occur.
- When generating code “Full separation” must be switched on. If no separation is used, this restriction will not occur.

If all three conditions are met there will be a compilation error from compilation of the <system>.c file. The function cm_Free_Signal requires all signal declarations to be able to free the parameters for the signals. When signals are declared on block level the header file for such blocks are not included in the compilation of the <system>.c file.

If these three conditions apply you must include the needed header files manually on system level, this can for example be done by adding a text symbol on system level:

```
/*#CODE
#HEADING
#include "x.h"
#include "y.h"
*/
```

Chapter

66 *The Cmicro Library*

The Cmicro Library consists of a configurable SDL kernel together with all the necessary SDL data handling functions. The collection of C functions and C modules make up the so called SDL machine.

The Cmicro Library is used in combination with the C code generated by the Cmicro SDL to C Compiler. For information on the generator see chapter 65, *The Cmicro SDL to C Compiler*.

The scaling facilities in Cmicro mean that it is useful for both micro controller and real-time applications. The Targeting Expert will help to scale and configure the generated code and the library. Please view chapter 59, *The Targeting Expert*.

The SDL Target Tester offers the ability to target tests and debug Cmicro code. Please see chapter 67, *The SDL Target Tester*.

Introduction

The Cmicro Library is required for handling the SDL objects that have been generated from SDL into C with the Cmicro SDL to C Compiler. This means that the C code which was generated with the Cmicro SDL to C Compiler cannot be used without the Cmicro Library. It also means that the C code generated with Cadvanced cannot be used together with the Cmicro Library and vice versa.

The Cmicro Library is a collection of C functions and C modules which consists of:

- The non preemptive Cmicro Kernel
- The preemptive Cmicro Kernel
- All functions which are necessary to handle SDL data
- Utility functions

Note:

The Cmicro preemptive kernel is only available if the according license is available.

Furthermore the Cmicro Library can be instrumented with SDL Target Tester functionality.

Before the Cmicro Library can be used, some adaptations and configurations must be made. The Targeting Expert is a tool which helps in configuring the application, node, component and the SDL Target Tester.

In this chapter, the C source code of the Cmicro Library and the generated C code of an application are described. Although a separate chapter is dedicated to the SDL Target Tester (see chapter 67, *The SDL Target Tester*), a few features are outlined here. The following topics are discussed:

1. The section "Differences between Cmicro and Cadvanced" on page 3456 helps in taking care of compatibility between different C code generators. There are things which must be observed in SDL diagrams.
2. The section "The SDL Scheduler Concepts" on page 3460 gives information about how the SDL scheduler works. This is also of interest for creating SDL specifications with the most highest conformity because the different SDL schedulers work differently.

3. The section "Targeting using the Cmicro Package" on page 3481 presents all the steps that must be carried out for doing targeting with Cmicro.

4. The section "Compilation Flags" on page 3486 presents a complete list of all the flags that the Targeting Expert recognizes. The different flags are explained in detail.

5. The section "Adaptation to Compilers" on page 3523 must usually be read before the C code generated by the Cmicro SDL to C Compiler and Cmicro Library can be compiled to form an executable. This section outlines the steps that must be carried out for introducing a C compiler which is not in the available list. The Targeting Expert offers an easy to use feature to add a new compiler. Please view "Compiler Definition for Compilation" on page 2910 in chapter 59, *The Targeting Expert*.

6. The section "Bare Integration" on page 3530 gives detailed information on how to adapt the generated SDL system to the environment. Communication with handwritten C code, as well as communication between SDL and the operating system or a naked machine, is described here.

7. The section "File Structure" on page 3565 explains the functionality of the different C files delivered with Cmicro.

8. The section "Functions of the Basic Cmicro Kernel" on page 3570 contains a list of C functions that are usually included in a target executable plus a short description.

9. The section "Functions of the Expanded Cmicro Kernel" on page 3592 contains a list of C functions that are usually not included in a target executable. The functions listed in this section can usually be left out because they have meaning for special forms of integration in target systems only. The functions include a short description.

10. The section "Technical Details for Memory Estimations" on page 3598 presents some information to the user which enables him to roughly estimate the consumption of memory in the target system. A self defined benchmark test is included, too.

Pay extra attention to the subsection "Automatic Scaling Included in Cmicro" on page 3519, which contains important information on the differences to the Cadvanced Library.

If examples of generated code are given, they are always shown in ANSI style. In addition, only the important parts of the code are shown to increase readability in the examples.

Differences between Cmicro and Cadvanced

General

This description deals with the differences between the generated C code of the Cadvanced SDL to C Compiler and the Cmicro SDL to C Compiler and the run-time libraries. There are differences because the main application area for the code generators differ.

Some of the differences discussed in the following are of interest for SDL users, while others are not.

SDL Restrictions

The Cmicro SDL to C Compiler has more SDL restrictions than the Cadvanced SDL to C Compiler. The additional restrictions are described in the subsection "SDL Restrictions" on page 3450 in chapter 65, *The Cmicro SDL to C Compiler*.

Furthermore, there are restrictions within the Cmicro Library that may affect the user's SDL system design. These restrictions are listed in the following. More technical information can be found in the section "Generation of Identifiers" on page 3446 in chapter 65, *The Cmicro SDL to C Compiler*.

- The run time model in Cmicro is such that there are global variables used in the generated C code and the Cmicro Library. The code generation uses "Auto initialization in C" quite extensively because the C compiler then can produce code that is more efficient than if initialization would take place within a generated C function (like `yInit`).
- The run-time model of the preemptive kernel requires function recursion from the C compiler.

Scheduling

First, it must be emphasized that both schedulers conform fully with SDL, although the schedulers introduce a different behavior.

Cadvanced uses a process ready queue together with signal queues in order to schedule processes. Cmicro does not use a process ready queue but all scheduling is derived from one physical queue. This physical queue represents all SDL process input ports. Logically, or seen from SDL, each SDL process has its own input port.

Another difference is the preemptive scheduler of Cmicro, if it is used.

Thus, different execution of processes will occur.

It is also a question of SDL design whether the differences between Cadvanced and Cmicro can be externally recognized.

If no SDL process assumes a particular real-time behavior from its communicating partner process then the behavior – as seen from a black box view – is always the required one.

Generation of Files

The Cmicro SDL to C Compiler generates some more files. This is of interest when implementing build scripts, makefiles and so on.

It is important that after each change in the SDL system the Cmicro Library is re-compiled. The reason for this is the automatic scaling facility. Please view "Automatic Scaling Included in Cmicro" on page 3519.

Environment Handling Functions

The main differences arise when considering the environment handling functions. However, this only has consequences if the SDL environment is the same in both cases (if Cadvanced is used for simulation only and Cmicro for targeting, then the environment functions are to be implemented twice in any case).

Instead of including `scttypes.h` as in Cadvanced, for Cmicro the following include statements are to be introduced in the header of the environment module:

```
#include "ml_typ.h"
#include "<systemname>.ifc"
```

In general (for Cmicro as well as Cadvanced) four C functions are used to represent the SDL environment, namely

- `xInitEnv`,
- `xInEnv`,
- `xOutEnv`,
- `xCloseEnv`.

The functions have the same general meaning for Cadvanced and Cmicro, but there are a few differences so that it is necessary to implement the environment twice.

For Cmicro, each of the above C functions is compiled only if it is required (selected by `XMK_USE_xInitEnv, XMK_USE_xInEnv`...)

Differences occur in the declaration of the C functions `xInEnv()` and `xOutEnv()`.

The `xInEnv()` function of Cmicro carries no parameters.

The `xOutEnv`() function of Cmicro carries a few parameters which represent the signal which is to be output to the environment, including the signal parameters. It is recommended that the definition of the C function `xOutEnv()` should be written twice, once for Cadvanced and once for Cmicro.

The environment functions can also be generated with the help of the Targeting Expert.

Another difference is that signals and processes are identified in different ways. Cmicro does not use identifications like `xIdNode`. Instead, it uses fixed C defines to identify signals and processes. This is beneficial in reducing the amount of memory but has the consequence that each access to any signal and any process is to be implemented differently. Process type IDs are generated into the file `sdl_cfg.h`, signal IDs and type definitions are generated into the `<systemname>.ifc` file. The chapter about the Cmicro SDL to C Compiler gives more details on how identifiers are generated and can be used.

Including C Code in SDL by User

C code may be included in SDL by the user in the following cases:

- In order to connect SDL to the environment in a way other than via the C functions `xInEnv() / xOutEnv()`

- To use C constructs, which do not exist in SDL
- To use existing C code (e.g. C library functions)
- To implement the body of ADTs
- In an SDL task.

If any C code or C identifiers are used, then users must use the right identifiers and functions.

Generated C Code

Of course, the generated C code is different when comparing Cmicro with Cadvanced. It would take too much room to list all the details in this subsection. In any case these differences are of interest for certain technical reasons only and not for pure SDL users.

To compare the different code generator outputs, the user should refer to the subsection "Output of Code Generation" on page 3418 in chapter 65, *The Cmicro SDL to C Compiler*.

General Recommendations Regarding Compatibility

In order to reach full compatibility between Cadvanced and Cmicro, the following general recommendations should be followed:

- In general, C code should not be used in SDL diagrams.
- If there is no option other than to use C code, the C code should be written as compatible as possible, i.e. the C code should be written without using any C code generator or compiler specific commands. Also the C code should be placed at dedicated locations, and should not be distributed over the SDL diagrams.
- If it is not possible to reach full compatibility, then the C code must be written twice. A switch, which is used when invoking the C compiler, selects C code either for Cadvanced or for Cmicro. The macro `XSCT_CMICRO` will help to distinguish automatically.

The SDL Scheduler Concepts

In this section, the concepts of the Cmicro SDL scheduler are outlined, with particular emphasis on basic SDL, the handling of the queue, scheduling, signals, timers, states etc. To obtain information about data in SDL, especially ADTs, please consult chapter 56, *The Cadvanced/Cbasic SDL to C Compiler*. Here both predefined and user defined ADTs are outlined.

Signals, Timers and Start-Up Signals

Data Structure for Signals and Timers

Each signal that is either an ordinary SDL signal, a timer or the start-up signal used for the dynamic process creation, is represented by three structures:

- A C structure defining the entries in the queue
- A C structure defining the header of each signal
- A generated C structure representing the parameters of the signal

The first and the second is defined in `ml_typ.h`, the third is defined in the generated code as `yPDef_SignalName`. Some structure components are conditionally compiled, which is used to scale the system. Please view the following C structure:

```
typedef struct
{
   #ifdef XMK_USE_RECEIVER_PID_IN_SIGNAL
       xPID          rec;   /* Receiver process */
   #endif

   xmk_T_SIGNAL  signal;    /* Signalcode       */

   #ifdef XMK_USE_SIGNAL_PRIORITIES
      xmk_T_PRIO  prio;     /* Priority         */
   #endif

   #ifdef XMK_USE_SIGNAL_TIME_STAMP
      xmk_T_TIME time_stamp;/* Timestamp        */
   #endif

   #ifdef XMK_USE_SENDER_PID_IN_SIGNAL
       xPID         send    /* Sender process   */
   #endif
```

```
    #ifdef XMK_USED_SIGNAL_WITH_PARAMS
      xmk_T_MESS_LENGTH mess_length;
      union
      {
         void          *ParPtr;
         #if (XMK_MSG_BORDER_LEN > 0)
           unsigned char ParCopy[XMK_MSG_BORDER_LEN];
         #endif
      } ParUnion;
    #endif

} xmk_T_MESSAGE;
```

- `signal:`
  represents a C constant which uniquely identifies the signal or timer across the complete system.

- `rec and send:`
  represents the PID values of the receiver and sender of the signal. See the subsection "Processes and Process IDs (PID)" on page 3446 in chapter 65, *The Cmicro SDL to C Compiler* for details. These are conditionally compiled.

- `prio`:
  is used to store the priority of the signal. This is specified by using the directive `#PRIO` in the SDL diagram. It is compiled only, if signals with priority are stipulated by the user.

- `timestamp:`
  is used to store a timestamp, which is set when the SDL signal is put into the queue (output time). This component is also conditionally compiled.

- `mess_length:`
  represents the amount of parameter bytes in the signal.

Three different cases are to be considered concerning signals with parameters:

- If the signal has no parameters, then `mess_length` is set to 0 and `ParPtr` is a `NULL` pointer.

- If the signal parameters are larger than XMK_MSG_BORDER_LEN (*) bytes, then `mess_length` is set to the number of bytes and `ParPtr` is undefined.

- If more than XMK_MSG_BORDER_LEN (*) bytes of parameters are to be transferred, then `mess_length` defines the amount of bytes used for the parameters and `ParPtr` points to a dynamically allocated area.

Note:

This margin of XMK_MSG_BORDER_LEN bytes can of course be modified by the user to prevent dynamic memory allocation for any signal in the queue, or in contrast, to always use dynamic memory allocation. See the file `ml_mcf.h` to modify this.

The second structure, which encapsulates the above one, is used to administrate the signals in the queue, as required by the FIFO handling and SDL save construct:

```
typedef struct _T_E_SIGNAL
{
  xmk_T_MESSAGE       Signal;
  struct _T_E_SIGNAL *next;

#ifdef XMK_USED_SAVE
  xmk_T_STATE         SaveState;
#endif /* XMK_USED_SAVE */
} T_E_SIGNAL ;
```

- `next:`
  is a pointer which refers to the next entry of the queue. This is used by the Cmicro Kernel to get the next signal after the current one has been worked on.

- `SaveState:`
  contains either a dummy state value XEMPTYSTATEID or the state in which the process saved the signal. This is necessary to compare when the signal is to be consumed after state changes in the receiver process. It is used only if save is used anywhere in the SDL system.

As already mentioned, the above structure is used for ordinary SDL signals, timers and the start-up signal for the dynamic process creation.

No differentiation is made between signals and timers, except that signals and timers have a different identification (signal in the `xmk_T_SIGNAL` structure).

When a process is to be created, a start-up signal is sent. The start-up signal is tagged by a special priority value and a special id.

Dynamic Memory Allocation

Dynamic memory allocation is in principle not necessary with Cmicro. There are SDL systems which can live without any dynamic memory allocation and there are SDL systems that require dynamic memory allocation from the user's point of view. The users should in general try to prevent any dynamic memory allocation due to the problems this introduces. Soon or later memory leaks will occur.

Cmicro offers its own dynamic memory allocator. Please view "Dynamic Memory Allocation" on page 3543 for getting more information on this.

In the following subsections, the exceptions for when Cmicro uses dynamic memory allocation are listed.

Signals and Signal Parameters

In order to cope with efficiency, dynamic memory allocation should not be done, whenever possible. Cmicro offers two principles of memory allocation for signal instances, namely:

1. Signal instances are allocated from a static memory pool only.

   The static memory is allocated during compilation. The pool's size is predefined with the `XMK_MAX_SIGNALS` macro. To enable this configuration, the macro `XMK_USE_STATIC_QUEUE_ONLY` is to be set. This principle has the disadvantage that if there is no free memory in the static memory pool, a fatal error occurs. The user is however given the possibility to react on this error situation because the `ErrorHandler` C function is called.

2. As another principle, it is possible to first take memory from the static memory pool. If no more memory is available in that pool, further signal instances are created from the dynamic memory pool.

> **Caution!**
>
> This configuration is not available when the preemptive scheduling mechanism is used.

To enable the configuration, the macro XMK_USE_STATIC_AND_DYNAMIC_QUEUE must be set. The static memory pool's size is still predefined with the XMK_MAX_SIGNALS macro.

When signal instances are allocated from the static memory pool, the allocation procedure is the most efficient.

When the second principle is used, the execution speed will of course dramatically slow down because dynamic memory allocation happens. This change in the behavior of the Cmicro Kernel may cause problems in a real-time environment and the user should keep this in mind.

The static memory pool above is implemented as a predefined array in C. The array is dimensioned with the XMK_MAX_SIGNALS macro.

Dynamic memory is requested with the xAlloc C function and released again with the xFree C function. The body of both these C functions must in any case be filled out appropriately by the user in all cases.

Signal parameters are to be allocated dynamically, if the amount of parameter bytes exceeds a predefined constant. If the amount of signal parameters is below or equal this predefined constant, the parameter bytes are put into the signal's header.

The default value for this predefined constant XMK_MSG_BORDER_LEN is 4 bytes. Dynamic memory allocation only occurs if a signal carries more than XMK_MSG_BORDER_LEN bytes, where the latter is defined as 4.

If the user defines XMK_MSG_BORDER_LEN as 0, then for each signal, which has parameters, the C function xAlloc() is called.

If the user defines XMK_MSG_BORDER_LEN to 127, then

- for signals which have 127 parameter bytes or less, the parameters are copied into the xmk_T_SIGNAL structure.
- for signals which have more than 127 parameter bytes, the parameters are copied into an allocated area (using the C function xAlloc()).

Due to this mechanism, the allocation of signal parameters is also very fast. Releasing the memory is performed automatically by calling the xFree C function.

Predefined Sorts

Some of the predefined sorts require dynamic memory allocation. The sorts are

- charstring
- any ASN.1 sort that is based on charstring
- predefined Generators like Array (without a specified upper limit), String, Powerset, Bag, Ref

SDL Target Tester

If the SDL Target Tester is used, there are some more allocations from the dynamic memory pool. The SDL Target Tester allocates one block of dynamic memory in the start phase. This block is never de-allocated again. The size of the block depends on the amount of process types in the system. Another dynamic memory allocation takes place when a binary frame is to be sent to the host machine. The blocks that are allocated here are de-allocated again after the frame is put into the physical transmitter buffer.

Overview for Output and Input of Signals

Output and input of signals is performed according to the rules of SDL. To get detailed information about the implementation of output and input, please consult the subsection "Output and Input of Signals" on page 3476.

Signal instances are sent using the C function `xmk_Send()` or `xmk_SendSimple()`. The function takes a signal and puts it into the input port of the receiving process instance.

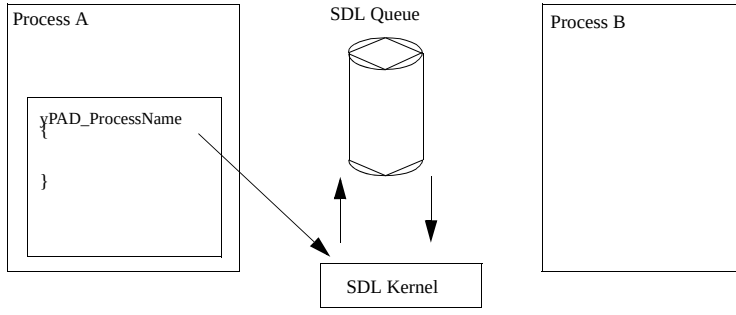


Figure 590: Queueing - sending side

There is no process ready queue. Physically, the queues of the different processes of the system are represented by just one queue. There are different principals to create signal instances which the user can choose between. The principals are explained within the subsection "Signals and Signal Parameters" on page 3463.

From an abstract point of view, it does not matter, if there is one physical queue for all the signal instances in the system, or if there is one physical queue for all the signal instances sent to one process instance. The fact that Cmicro uses one physical queue for all signals in the system only, has no effect on SDL users and conforms to the semantic rules of SDL. The scheduling simply depends on the ordering of signals in the queue. In the case of a preemptive Cmicro Kernel scaled this way, there is one linked list of signals per priority level all using the array mentioned above. See the subsection "Scheduling" on page 3469 for details.

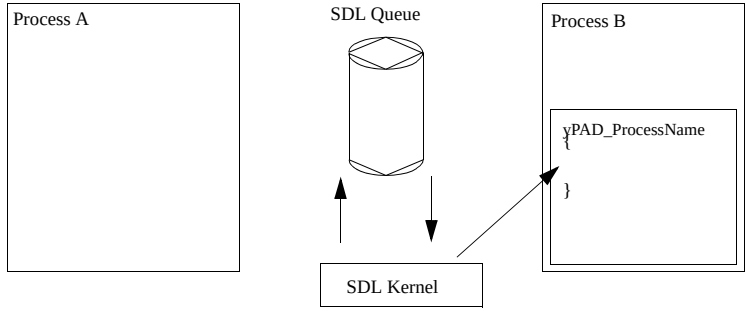


Figure 591: Queueing - receiving side

When working on a signal, the Cmicro Kernel decides between four different constellations:

- The receiving process instance is active. In its current state the signal is to be saved. The Cmicro Kernel tags the signal as "saved" and works on the next signal.
- The receiving process instance is active and there is a transition to be executed receiving the current signal. The Cmicro Kernel fires the transition.
- The receiving process instance is active but there is no transition to be executed for that signal in the current process state. The signal is in accordance with SDL rules implicitly consumed by the process.

- The receiving process instance is not active, i.e. either not yet created or already stopped. The signal will be discarded and the C function `ErrorHandler()`, discussed within the subsection "User Defined Actions for System Errors – the ErrorHandler" on page 3548, will be called.

In any case, except when being saved, the signal will be removed from the queue and returned to the list of free signals.

After performing the nextstate operation, the input port is scanned in accordance with the rules of SDL to find the next signal which would cause an implicit or explicit transition.

There is no specific input function. This functionality is contained in the Cmicro Kernel.

Timers and Operations on Timers

With the delivered timer model, all timer management entities fully conform to SDL. This means that more memory is required to implement this timer.

For each timer, there is a C structure defined in `ml_typ.h`.

```
typedef struct _TIMER
{
  struct _TIMER *next ;

  xmk_T_SIGNAL   Signal    ;
  xmk_T_TIME     time      ;
  xPID           OwnerProcessPID ;

} TIMER ;
```

`xmk_T_TIME` is defined as a long value.

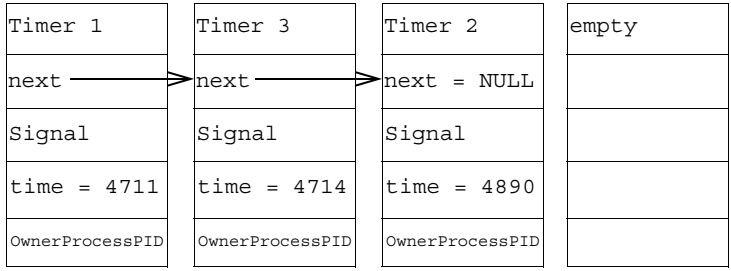


Figure 592: Handling of timers (timer model 1)

The component

- `next` is used to refer to the next entry;
- `signal` is the timer identification;
- `time` is used to store the absolute time when the timer should expire;
- `OwnerProcessPID` is used to refer to the process PID for which the timer was set.

As can be seen above, timers are implemented as a forward linked list.

- An array of timers is allocated during compile time
- The size of the array is calculated by the Cmicro SDL to C Compiler as the required maximum. It is defined by the macro MAX_SDL_TIMER_INSTS. This is also valid for multiple process instances, so that there is always enough memory to handle all timer instances which are possible. (If the user wishes to reduce the occupied RAM memory for timers, he must evaluate the maximum amount of timer instances required during run-time and modify the above define by hand).
- The timer which expires first is always put to the front of the linked list (increasing performance).
- The time value is stored as absolute time from the start of the SDL system.

Processes

Data Structure for Processes

Each process is represented by structures and tables containing SDL information, for example, tables containing transition definitions and a structure representing the variables of the process. The typedef for variables is generated in the generated code and named `yVDef_ProcessName`. For each process instance, there is one array element defined statically during compile time.

The structures and tables for processes are described in detail in the subsection "Tables for Processes" on page 3420 in chapter 65, *The Cmicro SDL to C Compiler*.

Scheduling

The Cmicro Kernel supports in principle the following scheduling policies:

- Non Preemptive Scheduling:
  Transitions in SDL cannot be interrupted.
- Preemptive Scheduling with Process Priorities:
  Transitions in SDL can be interrupted by any external or internal output. This could be a signal coming from the environment, i.e. an interface driver.
- In addition, signal priorities can be used to specify the ordering of signals in the signal queue. Signal priorities are discussed in each of the subsections mentioned above.

General Scheduling Rules

- All of the above policies basically operate on the SDL queue. Physically, the Cmicro Kernel uses one array for all process queues in the system. From the SDL point of view, the implementation conforms to SDL because each SDL process virtually has its own queue in the implementation.
- signal priorities can be used in all cases.
- process priorities are given priority treatment.

- If no signal priorities are assigned, then the ordering of signals depends on their arrival (FIFO strategy).
- In order to increase the performance, there is no process ready queue implemented. The scheduling of processes is fully derived from the SDL queue.

Further explanation of the scheduling is in the next subsection.

Non Preemptive Scheduling

The Cmicro Kernel takes the first signal from the queue and if the signal is not to be saved the appropriate SDL transition is executed until a nextstate operation is encountered. Outputs in SDL transitions as well as SDL create operations are represented by signals, where create signals are given priority treatment no matter whether signal priorities are used or not. This guarantees that there will not be any problems when a signal is sent to a process just before the process has been dynamically created.

The ordering of signals in the queue can be affected by using signal priorities (#PRIO directive for signals).

Whenever an output takes place, the Cmicro Kernel inserts the signal into the queue according to its priority. High priority signals are inserted at the queue's head, low priority signals at the queue's end. Create signals are still given priority treatment.

Note:

The priority of a create signal in the standard delivery is set to one.

This, as well as all the default signal priorities, is user-definable in the file `ml_mcf.h` (generated by the Targeting Expert). In this way it is possible to define signals with a higher priority then the create signal. The user should remember not to do so!

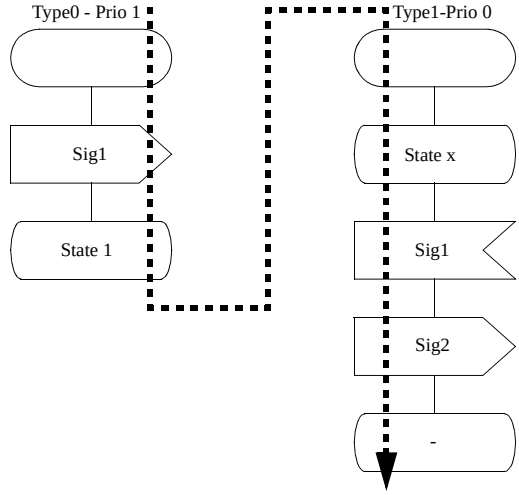


Figure 593: Non preemptive scheduling

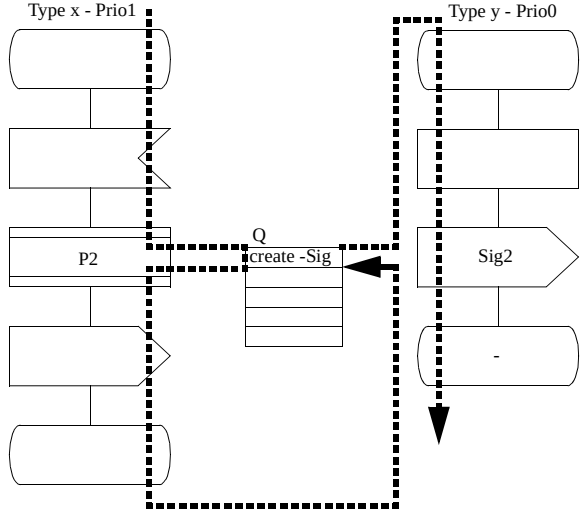


Figure 594: Scheduling for Create

A signal (this is either an ordinary SDL signal, a timer signal or the internal start-up signal in the event of SDL create) with the highest prior-

ity is always put in front of the SDL queue, a signal with the lowest priority is always put at the end of the SDL queue.

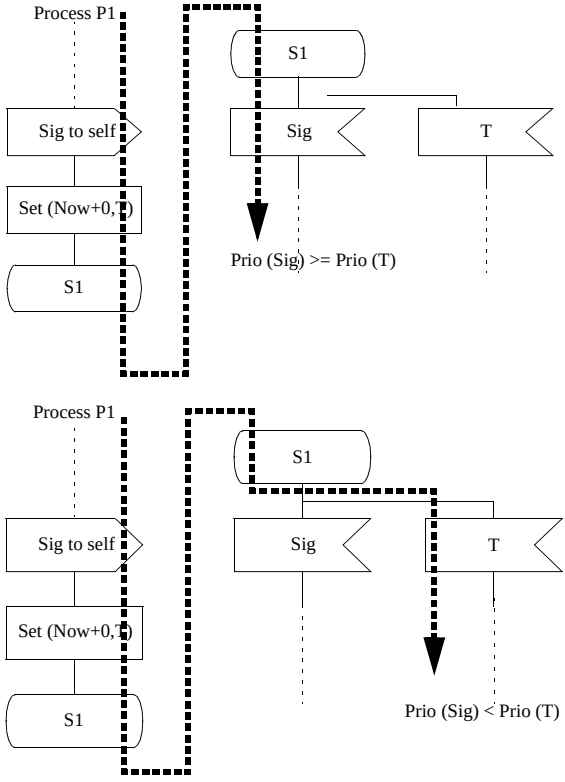


Figure 595: Signal priorities

For applications, which do not have time critical requirements, the non preemptive scheduling policy is the correct one to implement. The Cmicro Kernel memory requirements are also reduced when the non-preemptive scheduling policy is implemented. For example, for an interface with a very high transmission rate, the preemptive scheduling policy is better suited in order to increase the reaction time on external signals coming from the environment.

On starting the SDL system, processes are statically created according to their order of priority.

Preemptive Scheduling with Process Priorities

Note:

The Cmicro preemptive kernel is only available if the according license is available.

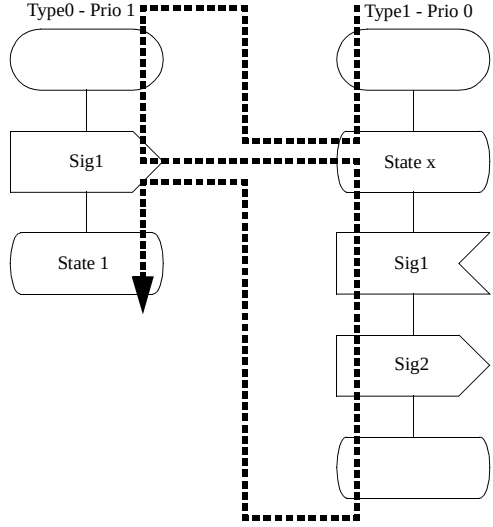


Figure 596: Preemptive scheduling

SDL assumes the start transitions of all statically created processes are already finished at system start-up (a transition takes no time according to SDL semantics). To simulate this, the Cmicro Kernel starts all statically created processes in the order of their priority before working on any signal. Preemption is disabled during the start-up phase of the system. There are two C functions namely `xmk_DisablePreemption` and `xmk_EnablePreemption` which can be used to prevent the Cmicro Kernel from performing a context switch. In this way it is possible to affect the scheduling from within the SDL system (using the #CODE directive).

The preemptive scheduling policy is absolutely necessary if an application consists of a mixture of processes, of which some have to react very quickly to external events, while others require enough time for pro-

cessing. The Cmicro Kernel does all things necessary to schedule such a mixture of processes.

Users only have to specify a high priority for processes which have to react after a short time. This can be done with the #PRIO directive for processes. If no `#PRIO` directive is specified, then a process is given the default value, which is specified by the user in the file `ml_mcf.h` (please view the subsection "Compilation Flags" on page 3486).

The highest priority is represented by the value zero. The numbering has to be consecutive with the priority decreasing with increasing numbers. Processes with the same priority are on the same priority level. The default process priority must be in the range of zero to the lowest priority value used in the system. Assume the following priority levels:

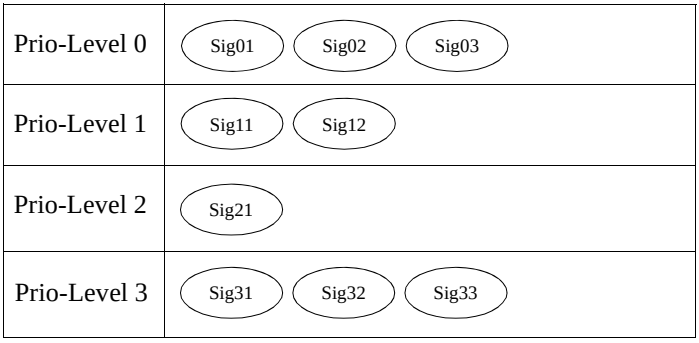


Figure 597: Priority levels

In the figure above there are signals queued for processes on four different priority levels. These would be worked on in this order:

According to their priorities the signals on priority level zero are consumed first, afterwards those of level one, two and last three. This is relevant only, if no signals are sent during the transitions executed because of the signals.

Rules

- At system start, SDL processes are statically created in accordance with their defined priority. During their start transitions it is possible for processes to send signals or create other processes but no preemption will take place until the start-up phase is completed, i.e. all

static processes have completed the transition from start state to first state.

- Processes running on a priority level never interrupt other processes running on the same priority level, thus two instances of the same type never run at one time.
- A higher priority level is entered, if a signal results in a process with a higher priority running.
- A lower priority level is entered, if all signals on the currently active priority level are consumed so that still no process is running on this priority level.
- Global variables are used to store some information concerning processes. The Cmicro Kernel is responsible for the storing and reloading of these, if a preemptive kernel is implemented.

Note:

If C variables are to be used in the SDL application, i.e. (x, N) declarations are used where N > 1 these variables are only available once and not once per process instance.

Restrictions

In order to produce portable C code, this version of the Cmicro Kernel uses recursive C function calls. A few C compilers available on the market do not support recursion. If such a C compiler is required, the user cannot use preemptive scheduling with process priorities.

Create and Stop Operations

No dynamic memory allocation occurs if an SDL create operation is performed. No freeing of memory occurs if an SDL stop operation is performed.

Data of a process instance is represented by the following typedef structs:

- A typedef struct representing PID values (parent and offspring) stored for an instance
- A typedef struct representing SDL states of an instance
- A typedef struct representing SDL data of an instance

The reason for using such a structure is to make it possible for the Cmicro Kernel to operate on the PID structure and the table with SDL states, so that no code needs to be generated in the application to update such variables.

An array of N elements for each of these typedef structs is generated, where N is the maximum number of process instances. The maximum number is to be specified in the process declaration header in the SDL diagram.

After performing a stop action, old PID values might exist in variables of other processes. The synchronization between processes to prevent situations where signals are sent to dead processes is left to the discretion of the user. If a process sends a signal to a non existent process, where nonexistent means either “never created” or “is dead”, the `ErrorHandler` is called and the signal is discarded (SDL conform).

When the Cmicro Kernel stops a process, the input queue assigned to the process stopped will be removed. No interpretation error occurs for the signals which existed in the queue before the process was stopped.

Note:

It is also possible for the user to implement dynamic memory allocation for process instance data. Some C defines are to be redefined in this case.

Output and Input of Signals

The actions to perform an output in the generated code are as follows:

- A struct variable is initialized with the type of the signal parameters, not using memory allocation.
- The struct variable is then filled, parameter per parameter in generated code.
- One of the `xmk_Send*` functions is called with a pointer to the location of the struct variable. There are several `xmk_Send*` functions in order to use the most effective one in SDL output.

Within the `xmk_Send*`- functions there are a few checks performed. For example, if the receiver is a NULL-PID, then the ErrorHandler is called.

Next, the environment C function `xOutEnv` is called. This function is to be filled out by the user. The function decides if the signal should be sent to the environment. The information necessary can be extracted from the signal ID, the priority or the receiver of the signal. If `xOutEnv` has "consumed" the signal, `xmk_Send*` returns immediately. `xOutEnv` has to copy the parameters of the signal to the receiver of the signal in the environment because after returning, the parameters will no longer exist.

If the signal is not environment bound, then the signal is sent to an internal SDL system process and `xmk_Send*` inserts the signal into the queue. This is done according to the priority of the signal (see subsection "Assigning Priorities – Directive #PRIO" on page 3415 in chapter 65, *The Cmicro SDL to C Compiler*.

If the signal carries no parameters, or if the signal parameters are represented by less than or equal to `XMK_MSG_BORDER_LEN` bytes, no dynamic memory allocation occurs and possible parameters are transferred directly by copying them into the signal header.

If more than `XMK_MSG_BORDER_LEN` bytes parameters are to be transferred dynamic memory allocation occurs. A pointer in the signal header then points to this allocated memory area. Freeing is done after consumption of the signal at the receiver process after executing the next-state operation or after the signal was consumed by the environment.

In order to implement this strategy, each signal carries a field "data length" in its header, to detect if a pointer is transferred or a copy of the parameters.

Note:

There are several possibilities to send signals to the environment by not using `xOutEnv`. The user should have a look at the `#EXTSIG`, `#ALT` and `#TRANSFER` directives which can be used in SDL Output. The one most similar to SDL, however, is the one which simply uses the C function `xOutEnv`.

At the receiver's side, when the input operation is to be performed, it is checked if the signal is to be saved or discarded. In the case of save, the next signal contained in the queue is checked and worked on. If on the other hand the signal is to be discarded, in the case of an "implicit transition", i.e. no definition is present to handle that signal in the current

state, the signal is then deleted from the input port of the process and, using the SDL Target Tester, the user is notified.

Otherwise, the signal leads to the execution of the transition. After performing the nextstate operation, the signal is deleted from the input port of the process and scheduling continues according to the rules of the scheduler again.

In the case of a so scaled preemptive Cmicro Kernel, the signal remains active in the input port until the nextstate operation is executed, although other processes can interrupt the running one (preemption).

Nextstate Operation

The nextstate operation is implemented by a return (return value) statement in the generated code. Several return values are possible to control the action to be taken by the Cmicro Kernel. The PAD function representing the SDL process can return any of the following (PAD means "Process Activity Description").

The return-value either expresses

- An ordinary SDL state change, the value is a C constant representing the SDL state as a simple integer (generated as #define),
- no SDL state change, indicated by the value `SDL_DASH_STATE`,
- SDL process stop operation, indicated by the value `SDL_STOP`.

After recognizing the action to be taken, the Cmicro Kernel either writes the process state variable or stops the SDL process.

The signal which was just being worked on is deleted from the input port. This includes the freeing of the memory area allocated for the signal, if memory was previously allocated.

Decision and Task Operations

No action is to be performed by the Cmicro Kernel for decisions and tasks, except those to trace these actions for the SDL Target Tester.

For SDL decisions and SDL tasks, C assignments and C function calls are implemented. Function calls are used in the case of an ADT or where simple C data operations (i.e. =, >, >=, ==) cannot represent the SDL operation wanted.

Note:

Function calls, however, are not necessarily generated, i.e. if the user defines C macros/defines instead of C functions for an ADT. Please consult the subsection "Abstract Data Types" on page 3406 in chapter 65, *The Cmicro SDL to C Compiler*.

Procedures

There are some special topics regarding procedures in SDL. The most important are:

- The location of data and how to access data
- Scope and visibility rules
- Procedures with states
- Recursivity
- Who can call a procedure

Note that procedures with states and remote procedure calls are **not supported** within the Cmicro Package.

It is possible to use global procedures (SDL'92) which makes it possible to specify a procedure once, by allowing several processes to call it.

Recursion is allowed, but should be introduced only if an algorithm cannot be designed alternatively. In most cases, algorithms and recursivity are subjects for an ADT.

Procedures returning values are also implemented. The return value in SDL is mapped to a return value in C. Procedures not returning values are mapped to C functions returning void.

The remaining part is the location of procedure data and the access to procedure data. Here another restriction exists, namely that it is not possible to access data of the father procedure of a procedure without declaring this explicitly.

Data, which belongs to a process is always located as a global array in C (for x, N process declarations, where N is >1). Data which belongs to a procedure is always allocated on a C stack for the called procedure.

Procedure Calls

An SDL procedure call is implemented as a direct function call in C. Each formal parameter is passed as a C parameter to the function. Ac-

cess to global data of the calling process is possible only if the procedure is not a global procedure. The C code generated for a local procedure uses global C variables of the surrounding process.

Data which is declared locally within a procedure is also allocated on the C stack.

No dynamic memory allocation is performed as procedures with states are not handled.

The following example shows the mapping for procedures returning values.

Example 605: Procedure Call

```
TASK i := (call p(1)) + (call Q(i,k));
```

is translated to something like:

```
i = p(1) + Q(i,k);
```

Note:

The value returning procedure calls are transformed to C functions returning values.

An SDL procedure can be called more than once. No conflicts occur if using a preemptive Cmicro Kernel.

Procedure Body

For each SDL procedure, there is one C function generated. The Procedure body can contain the same SDL actions as the process body. The same code generation is performed with the exception of a few statements declaring temporary variables.

Within global procedures, no objects of the calling process can be used without declaring them via formal parameters. Another restriction is that each output in a global procedure must be specified with **to PID**.

Blocks, Channels and Signal Routes

No C code is generated for blocks, channels and signal routes, except the C comment, which tells the user the location of processes and procedures.

Targeting using the Cmicro Package

Directory Structure

At first it is necessary to know the directory structure the Cmicro Package is stored in.

On UNIX, the Cmicro Package is contained in `$sdtdir/cmicro`.

In Windows, the Cmicro Package is contained in `%SDTDIR%\cmicro`.

```
cmicro
   +-- include
   +-- kernel
   +-- mcod
   +-- template
         +-- commlink
   +-- tester
```

Furthermore all files, which are used when targeting Cmicro, are definitely stored in this directory tree. Otherwise the files are generated by the Cmicro SDL to C Compiler or by the Targeting Expert.

Prerequisites

Before starting the targeting with Cmicro it is necessary to have an SDL system designed and already tested with the help of the Simulator.

This means targeting begins when the first testing phase is finished.

All steps of targeting will be discussed in the following sub-sections.

Different Steps in the Work Flow

All the different operation steps listed here will completely supported by the Targeting Expert (see chapter 59, *The Targeting Expert*).

1. To start the Targeting Expert, select the entry *Targeting Expert* in the Organizer's *Generate* menu.
2. Select pre-defined integration settings or <*user defined*> for a new integration. See "Pre-defined Integration Settings" on page 2977 in chapter 59, *The Targeting Expert*.
   - select the Code Generator Cmicro (if not automatically done)
   - select the compiler (if not automatically done)

   The Cmicro SDL to C Compiler will automatically be invoked and generates the following files (assumed that separation is not selected):

sdl_cfg.h	the automatic configuration
<systemname>.c	the SDL system
<systemname>.ifc	the environment header file
<systemname>_gen.m	the sub-makefile (Targeting Expert)
<systemname>.xrf	the X-References (not used)
<systemname>.sym	the symbol file (SDL Target Tester)

3. Specify the compiler, linker and make settings (if not automatically done).

   Please view "Configure Compiler, Linker and Make" on page 2930 in chapter 59, *The Targeting Expert*.
4. Configure and scale the generated C code and the Cmicro Library.
   - For information about the compilation flags and their interdependencies please view "Compilation Flags" on page 3486.
   - For more details how to use the Targeting Expert please view "Configure and Scale the Target Library" on page 2946 in chapter 59, *The Targeting Expert*.
5. Copy template files.

Several files of the Cmicro Package are delivered as template files. This is done because lots of things can only be done by the user as they must fit to the user's SDL system. The template files give easy to use C functions including help to adapt them to the user's needs.

All the files `mk_stim.c`, `mk_user.c` and `mk_cpu.c` should be copied into the project's directory tree (generated by the Targeting Expert). All these files are stored in the `template` directory.

If the SDL Target Tester should also be used it is necessary to copy the files `mg_dl.c` and the files describing the preferred communications link (e.g. the file `8051_v24.[ch]` if an 8051 micro controller and a V.24 interface should be used) into the project directory, too. The file `mg_dl.c` is stored in the `template` directory and the communications link files are stored in `template/commlink` if contained in the Cmicro Package delivery. Please view the sub-section "The Communications Link's Target Site" on page 3712 to get information about how to create an own communication link.

Please view also "Source Files" on page 2935 in chapter 59, *The Targeting Expert* to get information on how to add more files to the list of files to be compiled. The Targeting Expert will automatically add these files to the makefile.

6. Full make the complete system.

   It is probably necessary to do an environment connection first (see "Connecting the SDL Environment" on page 3483).

7. Download and execute the executable.

   The following sections in the Targeting Expert's manual (chapter 59, *The Targeting Expert*) should be checked to find out how these steps can be eased:

   – "Download Application" on page 2955

   – "Test Application" on page 2955 (which is probably the target executable itself)

Connecting the SDL Environment

From the Cmicro Package's point of view the SDL environment is represented in template C functions. A short overview of this C functions

is given here. For further information please view "Bare Integration" on page 3530.

- xInitEnv():
  This function is given as a template in the file `env.c`. It needs to be filled any time it is necessary to initialize specific hardware components on the target.

- xCloseEnv():
  `xCloseEnv()` is given as a template in `env.c`, too. It is only necessary to fill this function if the SDL system can perform a system stop.

- xInEnv():
  `xInEnv()` is basically generated by the Targeting Expert into the file `env.c`. This means that all the signals coming from the environment are already inserted into the C code of this function.

Note:

All the modifications of `env.c` should only be done between the comments `/* BEGIN User Code */` and `/* END User Code */`. All the other modifications will be lost when re-generating the `env.c` file.

- xOutEnv():
  For all the signals to the environment the Targeting Expert generates the C function `xOutEnv()` into the file `env.c`. All the signals to the environment are included into the C code.

Note:

All the modifications of `env.c` should only be done between the comments `/* BEGIN User Code */` and `/* END User Code */`. All the other modifications will be lost when re-generating the `env.c` file.

- User Defined Actions for System Errors – the ErrorHandler:
  The Cmicro Package supports the handling of detected errors by calling the C function `ErrorHandler()` in the module `mk_user.c`. The user needs to redirect the error messages to his target hardware. I.e. display the errors on the hardware.

The compilation and linkage of the environment functions can be prevented by using the flags given in "Compilation Flags" on page 3486.

Different Forms of Target Integration

Different forms of target integration, that is, integrating generated C code, are distinguished:

- Bare Integration: There is no operating system available on the target machine. The `main()` function of Cmicro can be used.
- Light Integration: An operating system is used and the complete SDL system executes in one operating system task. The SDL task communicates with the environment by using the communication resources of the operating system.
- System partitioning: An operating system is used, the SDL system may execute in different CPUs, or the processes of the SDL system execute in different OS tasks.
- Tight Integration: An operating system is used, the SDL system executes in one CPU, and the processes of the SDL system execute in different tasks.

Bare and Light integration represent the most easiest form of integration.

Bare integration is described in "Bare Integration" on page 3530.

Light integration is described in "Light Integration" on page 3559.

Tight integration can be performed with Cmicro, but the complexity makes it difficult to describe the integration in this section. It is a generic solution available, which guides thought the integration.

Please contact IBM Rational local sales office/Professional Services for further information.

Compilation Flags

Compilation flags are used to decide the properties of the Cmicro Library and the generated C code. Both in the Cmicro Library and in the generated code `#ifdef`'s are used to include or exclude parts of the code.

The switches used can be grouped:

1. Flags defining the compiler
2. Flags defining the properties of a compiler
3. Flags defining the properties of the Cmicro Library
4. Flags defining the implementation of a property

The first two groups are discussed in "Adaptation to Compilers" on page 3523.

The remaining two groups are discussed under "Manual Scaling" on page 3486 and "Automatic Scaling Included in Cmicro" on page 3519.

Flag naming conventions:

- `XMK_ADD_`: include optional parts
- `XMK_USED_`: automatically include parts of the Cmicro SDL to C Compiler. Note: Do not switch off by hand.
- `XMK_USE_`: manual scalings
- `XMK_MAX_`: manual scalings

Use the default settings of these flags in order to reduce potential problems.

Manual Scaling

Users are able to scale some features of the Cmicro Library and the Cmicro Kernel in order to optimize the generated code. All the flags discussed in this section are used throughout the complete SDL system, therefore it is not possible to define flags for processes separately.

> **Note:**
> The whole SDL system must always be generated.

The manual scalings have to be done in the file `ml_mcf.h`. They are divided into the following groups:

1. Cmicro Kernel/Library
   - Kernel
   - Signals
   - Timers
   - Error checks
2. Support of SDL Constructs
   - Predefined sorts
   - Size of variables
   - Use of memory
3. SDL Environment
4. SDL Coder
5. SDL Target Tester
   - Initialization
   - Trace
   - Record and Play
6. Communication Link and Compiler

In the following subsections, the way to configure the Cmicro Package by hand and all the available flags are described.

The user is asked to use the Targeting Expert to configure the Cmicro Package. A description of how to use the targeting Expert can be found in "Targeting Work Flow" on page 2926 in chapter 59, *The Targeting Expert*.

To configure the Cmicro Package by hand, the user has to modify the file `ml_mcf.h`.

To use a Cmicro Package feature just define it in `ml_mcf.h` like

```
#define FLAG_NAME_AS_DESCRIBED_BELOW
```

Sometimes a flag just carries a value. In this case the user has to modify the value in `ml_mcf.h`

```
#define FLAG_CARRYING_VALUE  VALUE
```

Cmicro Kernel/Library

Kernel

General

- XMK_USE_KERNEL_INIT

  When this flag is defined, the Cmicro Kernel `memsets` all variables of the SDL processes to 0 before the process is created. This is useful to spare some ROM memory but has the disadvantage of a longer process creation phase. The ROM used by the `yDef_SDL_*` functions can be spared in this case where the initialization with 0 is appropriate.

Initial setting	set

- XMK_USE_SDL_SYSTEM_STOP

  Normally, it is unnecessary to have an SDL system stop in micro controller applications. An SDL system stop occurs if no living process in the SDL system exists and all queues are empty. This occurs if all process instances have executed an SDL stop and no create process action is open (create signal in any queue).
  Defining `XMK_USE_SDL_SYSTEM_STOP` means that the C function `main` returns correctly with exit(...). To avoid this overhead in the Cmicro Kernel, the user should not define the above flag.

Default setting	unset

- XMK_SYSTEM_INFO

  If this flag is set, some functions available to get information about the SDL system during runtime. See "Functions to get System Information" on page 3593

Initial setting	unset

- XMK_USE_NO_AUTO_SCALING

  When this flag is defined, automatic scaling of SDL features is prevented. An example of flag usage is implementing a change to the SDL system which does not require re-compilation of the Cmicro Library. Note that if the number of processes, signals or timers is changed, the Cmicro Library must be re-compiled. The definitions contained in `sdl_cfg.h` are overwritten in `ml_typ.h`.

Initial setting	unset

- XMK_USE_KERNEL_WDTRIGGER

  If this flag is defined, the Cmicro Kernel calls the C function `WatchdogTrigger` before executing a transition. This function is present as a template in the Cmicro Library source files.

Initial setting	unset

- XMK_USE_INTERRUPT_SERVICE_ROUTINE

  If this flag is defined, the Cmicro Kernel will enable more critical paths in functions xmk_InsertSignal, xmk_AllocSignal and xmk_FreeSignal. This flag needs to be manually set if Interrupt Service Routine (ISR) is used and can lead to signal sending.

Initial setting	unset

Kernel Limits

- XMK_USE_HUGE_TRANSITIONTABLES

  Per default the Cmicro Package can handle up to 252 different transitions per process. If the SDL system exceeds this size, it is necessary to set this flag. See "Transition Table" on page 3424 for further information.

Initial setting	unset

- XMK_USE_MORE_THAN_250_SIGNALS

Per default the Cmicro Package is designed to handle up to 250 user defined signals. When using larger SDL systems it is necessary to define this flag. This results in larger RAM and ROM occupation and should only be done if necessary (see `xmk_T_SIGNAL`).

Initial setting	unset

- XMK_USE_MORE_THAN_256_INSTANCES

  Per default the Cmicro Package handle up to 256 instances per process type. If the system uses more than 256 instances per type it is necessary to set this flag.

Initial setting	unset

- XMK_USE_PAR_GREATER_THAN_250

  Per default the Cmicro Package is able to handle signal parameters up to a length of 250 octets. If greater signal parameters are used (at least once) this flag needs to be set. Please view `xmk_T_MESS_LENGTH` in `ml_typ.h` for further information.

Initial setting	unset

Signal Structure

- XMK_USE_SENDER_PID_IN_SIGNAL

  When this flag is defined a sender pid is included in each signal instance. It is possible to omit the sender pid if the system contains no to SENDER or to PARENT addressing.

Initial setting	set

- XMK_USE_RECEIVER_PID_IN_SIGNAL

  When this flag is defined a receiver pid is included in each signal instance. It is possible to omit the receiver pid if the user writes a C function `xRouteSignal()` which is given as a template in `mk_user.c`. Each signal type is then mapped to a unique receiver. It is recommended to define this flag in small systems where unique receivers exist for each signal type. It is important to note that in the case of dynamically created processes an internal create signal is used. If there are any (x, N) where N > 1, declarations in the system, the create signal cannot contain the receiver pid. The receiver pid is necessary for correct creation of processes. Never leave it out when using dynamic process creation.

Initial setting	set

- XMK_MSG_BORDER_LEN

  The value of this macro gives the length of signal parameters carried inside the signal. Note: If the parameter length exceeds this value a memory allocation is performed and the signal parameters are copied into this buffer. The pointer to the allocated buffer is carried inside the signal's structure (see `ml_typ.h` for the typedef `xmk_T_SIGNAL`).

Initial value	4

Signal Handling

- XMK_USE_xmk_SendSimple

  This flag enables the output C function `xmk_SendSimple()` defined in the Cmicro Kernel. If the SDL system contains several signals without parameters and priorities, it is useful to set this flag in order to select the more optimal output C function, `xmk_SendSimple()`, in the Cmicro Kernel.

Initial setting	set
Reset	XMK_USE_SIGNAL_PRIORITIES

- XMK_USE_SAFE_ADDRESSING

  Setting this flag includes a special entry in the signal queue which ensures the handling of sender and offspring in the start transition.

Initial setting	unset

Signal Queue

- XMK_USE_STATIC_QUEUE_ONLY

  Create signals only from the static memory. The static memory is predefined with XMK_MAX_SIGNALS.

Initial setting	set

Reset	XMK_USE_STATIC_AND_DYNAMIC_QUEUE

- XMK_USE_STATIC_AND_DYNAMIC_QUEUE

  Create signals from the static memory which is predefined with XMK_MAX_SIGNALS. If more signals have to be inserted memory is allocated from the dynamic memory pool.

Initial setting	unset
Reset	XMK_USE_STATIC_QUEUE_ONLY

- XMK_MAX_SIGNALS

  In the Cmicro Package, the SDL queue is physically implemented as one queue for all processes. The define discussed in this section represents the maximum amount of signals in the static signal instance memory pool (see "Dynamic Memory Allocation" on page 3543 for more information). It may be difficult to evaluate the maximum entries required during run-time because this totally depends on how the SDL system is specified, and target hardware performance. It is for example, impossible to state how much time hardware requires to process an SDL signal.
  By examining the SDL system it can be determined which processes have a long transition time and which processes send or receive more than one signal. Estimate by trying out worst case situations.
  A first estimation is also possible by calculating:
  maximum amount of process instances * 3
  For a more exact estimation the user should use the profiler contained in the SDL Target Tester to obtain the necessary information on how many entries are used during run time.
  Another method of helping to determine the maximum amount of signals required by the system is to use the exception handling mechanism offered by the `ErrorHandler`, i.e. when the queue is full and another signal is to be inserted then the `ErrorHandler` function is called.

Initial value	20

Note:

Using dynamic memory allocation does not prevent the user from estimating the required memory for the queue. Sooner or later problems arise (e.g. memory fragmentation) if dynamic memory management is used frequently. For this reason the Cmicro Package avoids where possible the use of dynamic memory management.

Light Integration

- XLI_LIGHT_INTEGRATION

  If this flag is set, some functionality is included to make a light integration easier.

Initial setting	unset
Automatic set	XMK_USE_INTERNAL_QUEUE_HANDLING XLI_LIGHT_INTEGRATION

- XMK_USE_INTERNAL_QUEUE_HANDLING

  This flag make additional functions for queues the handling available which are needed for light integrations.

Initial setting	unset
Reset	XLI_LIGHT_INTEGRATION

- XLI_INCLUDE

  The value of this is the name of the header file which is included in ml_typ.h when XLI_LIGHT_INTEGRATION is set. This file contains the definitions of the light integration macros. See "Define the macros which are needed in the task function." on page 3561

Initial setting	"li_os_def.h"

Tight Integration

The complexity of a tight integration is very high. But there is a generic solution available, which guides you through the integration.

Please contact IBM Rational local sales office/Professional Services for further information about this solution.

- XTI_USE_INTERNAL_OUTPUT

  For more information about this contact local Professional Services

Initial setting	unset

- XTI_USE_LOCK_IN_CREATE

  For more information about this contact local Professional Services

Initial setting	unset

Error Handler

- XMK_USE_MAX_ERR_CHECK

  When this flag is defined, additional error checks are included in the generated code of the Cmicro Library. For further details, see the appropriate section on “errors and warnings”. For example the Cmicro Kernel calls the ErrorHandler if a signal is sent to an undefined process (i.e. undefined pid value).

Initial setting	set
Reset	XMK_USE_MIN_ERR_CHECK XMK_USE_NO_ERR_CHECK

- XMK_USE_MIN_ERR_CHECK

  In comparison to `XMK_USE_MAX_ERR_CHECK` this flag only includes a basic set of error checks.

Initial setting	unset
Reset	XMK_USE_MAX_ERR_CHECK XMK_USE_NO_ERR_CHECK

- XMK_USE_NO_ERR_CHECK

  When this flag is defined, all error checks are excluded. It is recommended not to use this flag until the post testing phase of the implementation, where it can be reasonably assumed that no errors remain. For further details see the appropriate section on “errors and warnings”.

Initial setting	unset
Reset by	XMK_USE_MAX_ERR_CHECK XMK_USE_MIN_ERR_CHECK

- XMK_USE_MON

  If the target executable runs on an environment with monitor functions, the module `ml_mon.c` can be included to have access to functions like `xxmonhexasc()` etc.

Initial setting	unset

- XMK_ADD_PRINTF

  When this flag is defined some additional `printf` C function calls are compiled giving users more information about the internal work of the system. The `printf` function can be switched on separately for Cmicro Kernel, SDL Target Tester and SDL application. At a lower level, it can be switched on separately for each C module. Look at the defines in `ml_typ.h`, which are all named as `XMK_ADD_PRINTF_*`.
  This flag must be undefined when compiling for the target, except in the case where there is a `stdio` implemented on the target. For correct compilation, the user must also set XMK_ADD_STDIO. If the user wishes to implement user specific `printf` functionality then this flag need not be set.

Initial setting	unset
Automatic set	XMK_ADD_STDIO

Reactions on Warnings

- XMK_WARN_ACTION_HANG_UP

If this flag is defined, the default behavior, if a warning is detected during SDL execution, is defined as “program hang up”.

The user might choose this reaction when there is no output device (like `printf`) available in the SDL program environment.

The user should notice that a warning can lead to an illegal system behavior. As an example, this might come true for an implicit signal consumption. The system then hangs but the user might perhaps not be able to see the reason why this occurs. As a result, it is recommended to trace for warnings also.

Initial setting	set
Reset	XMK_WARN_ACTION_PRINTF XMK_WARN_ACTION_USER

- XMK_WARN_ACTION_PRINTF

  By defining this flag, the user chooses that in the case of a warning, a `printf` function call with an appropriate error text should occur. Please see `XMK_WARN_ACTION_HANGUP` also.

Initial setting	unset
Reset	XMK_WARN_ACTION_HANG_UP XMK_WARN_ACTION_USER

- XMK_WARN_ACTION_USER

  By defining this flag, the user chooses that in the case of a warning, a user defined function should be called. The user’s function name must then be specified with `XMK_WARN_USER_FUNCTION`. Please see `XMK_WARN_ACTION_HANGUP` also.

Initial setting	unset
Reset	XMK_WARN_ACTION_HANG_UP XMK_WARN_ACTION_PRINTF

- XMK_WARN_USER_FUNCTION

If the flag XMK_WARN_ACTION_USER is defined, then the user must define the name of a C function with this macro. Please see XMK_WARN_ACTION_HANGUP also.

Initial setting	user_function()

Reaction on Errors

- XMK_ERR_ACTION_HANG_UP

  If this flag is defined, the default behavior, if a fatal system error is detected during SDL execution, is defined as “program hang up”.

  The user might choose this reaction when there is no output device (like printf) available in the SDL program environment.

  The user should notice that if a fatal system error is ignored, this usually leads to an illegal system behavior. As an example, this might come true for any use of null pointer values, for which there is an error check. It is strongly recommended to trace for system errors and warnings.

Initial setting	set
Reset	XMK_ERR_ACTION_PRINTF XMK_ERR_ACTION_USER

- XMK_ERR_ACTION_PRINTF

  By defining this flag, the user chooses that in the case of a system error, a printf function call with an appropriate error text should occur. Please see XMK_ERR_ACTION_HANGUP also.

Initial setting	unset
Reset	XMK_ERR_ACTION_HANG_UP XMK_ERR_ACTION_USER

- XMK_ERR_ACTION_USER

  By defining this flag, the user chooses that in the case of an system error, a user defined function should be called. The user’s function name must then be specified with XMK_ERR_USER_FUNCTION. Please see XMK_ERR_ACTION_HANGUP also.

Initial setting	unset
Reset	XMK_ERR_ACTION_HANG_UP XMK_ERR_ACTION_PRINTF

- XMK_ERR_USER_FUNCTION

  If the flag `XMK_ERR_ACTION_USER` is defined, then the user must define the name of a C function with this macro. Please see `XMK_ERR_ACTION_HANGUP` also.

Initial setting	user_function()

Timer Scaling

If there are timers used within the SDL system, the timers have to be scaled.

- XMK_USE_TIMER_SCALE

  The timer units can be scaled within the Cmicro Kernel. This means that the factor given with XMK_USE_TIMER_SCALE_FACTOR is used.

Initial setting	unset

- XMK_USE_TIMER_SCALE_FACTOR

  The factor modifies the expiration of timers. Is a value of 1 given here, it means that the expiration time is un-modified. A value of 100 increase the expiration time with a factor of 100 with other words the timer needs 100 times longer.

Initial value	100

- XMK_TIMERPRIO

  An expired timer is handled like a signal inside the Cmicro Kernel and inserted into the signal queue. In this way there has to be a priority level if signal priorities are used. The XMK_TIMERPRIO gives the priority for all expired timers. This default value can lay between 0 and 250.

Initial value	100

Timer queue

- XMK_USE_GENERATED_AMOUNT_TIMER

  The code generator evaluates the amount of timers that are declared in the system. The result of this evaluation is then, after code generation, defined with `XMK_MAX_TIMER_INST` in the file `sdl_cfg.h`. Timer instances are, usually, implemented as a C array with Cmicro. If the `XMK_USE_GENERATED_AMOUNT_TIMER` flag is set, then the timer array is dimensioned with the evaluated amount (`XMK_MAX_TIMER_INST`).

  When timers with parameters are in the system, the automatically generated value `XMK_MAX_TIMER_INST` can be used to pre-define one timer instance of a timer declaration. If there are then more timers to be instantiated, dynamic memory allocation must take place. This cannot be evaluated by the code generator.

  As a result, if there are timers with parameters in the system, the user should think about how many timer instances there could be during execution and should decide upon the maximum amount by himself. In this way, memory consumption and performance can be balanced.

Initial value	set

- XMK_MAX_TIMER_USER

  This macro is used for internal purposes. The meaning of it is the opposite of `XMK_USE_GENERATED_AMOUNT_TIMER`.

  The flag is set when `XMK_USE_GENERATED_AMOUNT_TIMER` is not set. The flag `XMK_MAX_TIMER_USER` is unset if `XMK_USE_GENERATED_AMOUNT_TIMER` is set.

Initial value	unset

- XMK_MAX_TIMER_USER_VALUE

  If `XMK_MAX_TIMER_USER` is set, the value `XMK_MAX_TIMER_USER_VALUE` becomes meaningful. With this val-

ue, the user may specify how many timer instances should be predefined during compilation time. The predefined timer instances offer the advantage that no dynamic memory allocation is to be done for them. If the predefined amount of timer instances is exceeded, then dynamic memory allocation will occur. As a result, it is up to the user how he balances static and dynamic memory for timer instances by changing this value.

Initial value	20

Execution Time

- XMK_USE_CHECK_TRANS_TIME

  When this flag is defined the time duration for each executed transition is checked against a predefined duration. If the executed duration is longer than the predefined duration set by the `XMK_TRANS_TIME` flag, the `ErrorHandler()` is called. The flag is only available when using a non preemptive Cmicro Kernel. No additional hardware is necessary as the evaluation is based on the SDL time value `NOW` which is provided by the same hardware source as for the system clock.

Initial setting	unset
Automatic set	XMK_TRANS_TIME

- XMK_TRANS_TIME

  This macro specifies the time duration that a transition execution time is compared with. The value must be in the target system time units, e.g. if the target's system step is 0,002 sec, the value 100 specifies a duration of 0.2 sec.

Initial value	20
Depend on	XMK_USE_CHECK_TRANS_TIME

Signal Priorities

In the standard configuration of the Cmicro Kernel no signal priorities are used. So each signal sent is inserted at the end of the signal queue.

- XMK_USE_SIGNAL_PRIORITIES

When this flag is defined, the header of each signal which is sent via the SDL queue has an additional entry, namely “priority”. This is used to modify the ordering of signals in the queue.
This flag should be used in combination with using the #PRIO directive. It works for all the different scheduling methods.
If signal priorities are enabled, signal handling `xmk_SendSimple` has to be switched of.

Initial setting	unset
Automatic set	xDefaultPrioSignal XMK_CREATE_PRIO
Reset	XMK_USE_xmk_SendSimple

- xDefaultPrioSignal

  If a signal has not got a priority assignment within the SDL system, it is handled with the default value `xDefaultPrioSignal`. A value between 0 (highest priority!) and 250 should be entered.

Initial value	100
Depend on	XMK_USE_SIGNAL_PRIORITIES

- XMK_CREATE_PRIO

  Create signals can not be assigned a signal priority within the SDL system. As these signals are also handled over the signal queue, a default priority is necessary when signal priorities are selected (i.e. `XMK_USE_SIGNAL_PRIORITIES` is defined). A value between 0 (highest priority!) and 250 should be entered.

Initial value	1
Depend on	XMK_USE_SIGNAL_PRIORITIES

Preemption

Normally, the Cmicro Kernel is configured so that the simple scheduling policy is used, i.e. transitions of SDL processes are non interruptible.

Note:

The Cmicro preemptive kernel is only available if an according license is available.

- XMK_USE_PREEMPTIVE

  Usually the Cmicro Kernel is configured to “non preemptive”. This means, that each transition must execute to its end, before the next transition can be processed.
  This means, that the nextstate symbol of a process is executed, before the next SDL input can be handled. On the other hand, “preemptive scheduling” is useful, if an SDL process must directly execute when an external event from an interrupt source is detected and sent to the SDL system. In this case, an executing transition is possibly suspended, and another transition can be executed.
  The kernel makes decisions based on the defined priority of processes when to schedule to another process. If the receiver of a signal which has been received from the environment (or a process inside the SDL system) has a higher priority than the currently executing process, the new signal is treated immediately.

Initial setting	unset
Automatic set	MAX_PRIO_LEVELS xDefaultPrioProcess XMK_USE_RECEIVER_PID_IN_SIGNAL
Reset	XMK_USE_CHECK_TRANS_TIME

- MAX_PRIO_LEVELS

  This defines the amount of process priority levels in the SDL system. It is of relevance only, if preemption is selected and must be equal to the lowest process priority plus 1. The lowest priority has the highest value.

Initial value	1
Depend on	XMK_USE_PREEMPTIVE

Caution!

A run-time error occurs if this definition is wrong. If the error checks are enabled, a check is made by the Cmicro Kernel and the C function `ErrorHandler()` is called if the check fails.

- xDefaultPrioProcess

  If there is an SDL process with no process priority assigned to it, this process will be handled with the `xDefaultPrioProcess`. It is mandatory that the `xDefaultPrioProcess` is in the range of 0 to (`MAX_PRIO_LEVELS` -1).

Initial value	0
Depend on	XMK_USE_PREEMPTIVE

SDL Target Tester

Initialization

- XMK_ADD_MICRO_TESTER

  This is the main flag to enable or disable the SDL Target Tester. Use this flag if the addition of the SDL Target Tester features is wanted. The following features are selected by this flag:

  – Trace into a file.
  – Trace into a buffer on target side.
  – Trace via a serial interface.
  – the Cmicro Recorder

Initial setting	unset
Automatic set	XMK_USE_MAX_ERR_CHECK

Reset when unset	XMK_ADD_MICRO_COMMAND, XMK_USE_DEBUGGING, XMK_ADD_PROFILE, XMK_ADD_MICRO_ENVIRONMENT, XMK_WAIT_ON_HOST, XMK_ADD_SDLE_TRACE, XMK_ADD_SIGNAL_FILTER, XMK_USE_COMMLINK, XMK_ADD_MICRO_TRACER, XMK_ADD_TEST_OPTIONS, XMK_USE_SIGNAL_TIME_STAMP, XMK_ADD_MICRO_RECORDER

- XMK_WAIT_ON_HOST

  During the start up of the target executable (see `xmk_MicroTesterInit()` in the module `mk_main.c`), the target can wait for the host's run-time configuration. If this is not needed, the `XMK_WAIT_ON_HOST` flag has to be undefined.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

- XMK_ADD_PROFILE

  This flag is to be defined if the profiler is to be used. With the profiler, it is possible:

  – to trace the SDL queue concerning it's traffic load,
  – to trace the number of timers which exist in parallel (not included in this version of the Cmicro Package).
  – to trace the execution time process transitions

  The profiler can be used independently from the SDL Target Tester functions. By using a debugger the following values may be inspected:

  `int xmk_max_q_cnt,`

  representing queue-dimensioning, and

  `int xmk_act_q_cnt,`

  representing the maximum traffic load of the queue at any time during the execution.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

- XMK_USE_DEBUGGING

  If the debugging functions should be included to the target executable, this flag needs to be defined.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

- XMK_ADD_MICRO_COMMAND

  This flag adds the command interface of the SDL Target Tester (please view chapter 67, *The SDL Target Tester*).

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

- XMK_ADD_MICRO_ENVIRONMENT

  To allow the signal exchange with the external environment connected to the SDL Target Tester, this flag needs to be defined.

  The external environment can be a GUI application or a cmdtool for example.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

Trace

Trace Scaling

- XMK_ADD_MICRO_TRACER

  This flag enables the trace of the SDL execution and trace of system events to any device.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER
Reset when unset	XMK_USE_SIGNAL_TIME_STAMP

- XMK_ADD_SDLE_TRACE

  This flag enables the trace of SDL symbols in the SDL Editor.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

- XMK_USE_COMMLINK

  If this flag is defined the trace via the specified communications link is enabled. The trace is handled with the SDL Target Tester's data link layer. See the module `mg_dl.c` for further information.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

- XMK_USE_AUTO_MCOD

  Is this flag defined, the host get a further message for the message coder. Depending on this flag the Targeting Expert insert the entry 'USE_AUTO_MCOD yes' into the sdtmt.opt file. The entrys LENGTH_, ENDIAN_, and ALIGN_ are ignored. For more information view chapter 67, *The SDL Target Tester*.

Initial setting	set

- XMK_ADD_SIGNAL_FILTER

  This flag conditionally compiles the signal filtering mechanism of the SDL Target Tester (please view chapter 67, *The SDL Target Tester*). With the signal filter, it is possible to specify that some signals should be traced while others are not.

Initial setting	unset

Depend on	XMK_ADD_MICRO_TESTER

- XMK_ADD_TEST_OPTIONS

  By defining this flag the symbol trace of the target can be configured. See function `main()` in module `mk_user.c`.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER

- XMK_USE_SIGNAL_TIME_STAMP

  Defining this flag means that in the header of each signal which is sent via the SDL queue, there is an additional entry “time-stamp”. When using the standard Cmicro Library, the Cmicro Kernel sets the time-stamp to `NOW`, if an SDL signal is inserted into the queue.

  This time-stamp is especially used by the Cmicro Recorder (XMK_ADD_MICRO_RECORDER) in order to enable the replay of an SDL session in simulated real-time.

  In other cases, the user is free to modify the type of time-stamp implementation.

Initial setting	unset
Depend on	XMK_ADD_MICRO_TESTER
Reset when unset	XMK_ADD_REALTIME_PLAY

- XMK_MAX_PRINT_STRING

  The user can add his own trace messages to the standard trace of the SDL Target Tester. This message will be handled like a string. The maximum size of these strings are defined with `XMK_MAX_PRINT_STRING`.

Initial value	40

Buffers

- XMK_MAX_RECEIVE_ENTRIES

If the ring buffer is used this flag gives the amount of entries for the receiver buffer.

Initial value	20

- XMK_MAX_RECEIVE_ONE_ENTRY

  This flag gives the size of one entry of the receiver buffer. The message received here are SDL Target Tester commands or the Cmicro Recorder's play data. I.e. as a rule over the thumb a value of 12 + maximum signal parameter size must be given here.

Initial value	100

- XMK_MAX_SEND_ENTRIES

  When the Cmicro Tracer is used the target executable writes the trace data into a ring buffer (view `ml_buf.c`). The recommended amount of entries given here depends of the used communications link. I.e. if using a slow communications link like V.24 this amount should be increased.

Initial value	20

- XMK_MAX_SEND_ONE_ENTRY

  When the Cmicro Tracer is used the target executable writes the trace data into a ring buffer (view `ml_buf.c`). The size of one entry depends on the SDL system. As a rule over the thumb a value of 12 + maximum signal parameter size must be given here.

Initial value	100

Recorder and Play

Note:

The SDL Target Tester's Record and Play functions are only available if a Cmicro Recorder license is available.

- XMK_ADD_MICRO_RECORDER

This flag adds the functions of the record mode of the Cmicro Recorder (please view chapter 67, *The SDL Target Tester*).

Initial setting	unset
Depend on	XMK_ADD_MICRO_TRACER
Reset when unset	XMK_ADD_REALTIME_PLAY

- XMK_ADD_REALTIME_PLAY

  When defining this flag, the replay of a recorded session can be performed in simulated real-time.

Initial setting	unset
Automatic set	XMK_USE_SIGNAL_TIME_STAMP
Depend on	XMK_ADD_MICRO_RECORDER

Support of SDL Constructs

Predefined Sorts

Character Strings

- XRESTUSEOFCHARSTRING

  This flag must be seen in combination with XNOUSEOFCHARSTRING, which forbids the use of SDL charstrings. If this flag is set, SDL charstrings are only supported in a restricted way. I.e. constant character buffers are used in this implementation.

  As a third possible way XRESTUSEOFCHARSTRING and XNOUSEOFCHARSTRING should both be undefined to enable a full support of SDL charstrings.

Initial setting	set
Reset by	XNOUSEOFCHARSTRING

Caution!

The Cmicro Recorder is not able to handle SDL charstrings within signal parameters.

- XNOUSEOFCHARSTRING

  The use of SDL charstrings is not possible. This can be selected for systems witch do not use SDL charstrings.

Initial setting	set
Reset by	XRESTUSEOFCHARSTRING

Predefined Sorts

- XNO_LONG_MACROS

  The setting of this flag prevents the use of predefined generators (please view "sctpredg.h, sctpred.h and sctpred.c" on page 3566). This can be done to reduce the target's memory size. But must be done for some specific compilers as these are not able to handle long macro definitions. Please view the appropriate compiler section in `ml_typ.h` ("Adaptation to Compilers" on page 3523)

Initial setting	set

- XNOUSEOFREAL

  This flag allows using real values in SDL if it is undefined.

Initial setting	set

Note:

Every multiplication of an integer value with an integer value is mapped to a multiplication of a real value with a real value. This is done because an overflow cannot be detected when using integer values.

ASN.1 Sorts

- XNOUSEOFASN1

If this flag is undefined it is allowed to use ASN.1 data types. It is recommended not to use ASN.1 data types if possible, because the functions that are necessary to handle these types occupy much memory.

Initial setting	set
Depend on	XNOUSEOFCHARSTRING

- XNOUSEOFOCTETBITSTRING

  If this flag is undefined it is allowed the use of ASN.1 data type octetbitstring. It is recommended not to use ASN.1 data types if possible, because the functions that are necessary to handle these types occupy much memory.

Initial setting	set
Depend on	XNOUSEOFASN1

- XNOUSEOFOBJECTIDENTIFIER

  If this flag is undefined it is allowed the use of ASN.1 data type objectidentifier. It is recommended not to use ASN.1 data types if possible, because the functions that are necessary to handle these types occupy much memory.

Initial setting	set
Depend on	XNOUSEOFASN1

Error Checks

- XEREALDIV

  If this flag is defined, then a C function is used to perform division of real values, otherwise a C macro is used. The flag should be set for allowing better error checks, but may be unset for optimizing the target program code, as the checks occupy a lot of memory.

Initial setting	set

- XEINTDIV

If this flag is defined, then a C function is used to perform division of integer and octet values, otherwise a C macro is used. The flag should be set for allowing better error checks, but may be unset for optimizing the target program code, as the checks occupy a lot of memory.

Initial setting	set

- XEINDEX

  If this flag is defined, then additional error checks are introduced that check the index of array at each position an array element is accessed. The flag should be set for allowing better error checks, but may be unset for optimizing the target program code, as the checks occupy a lot of memory.

Initial setting	set

- XEFIXOF

  If this flag is defined, then additional error checks are introduced that check the conversion from real to integer. The flag should be set for allowing better error checks, but may be unset for optimizing the target program code, as the checks occupy a lot of memory.

Initial setting	set

- XECSOP

  Setting this flag includes an error check for predefined ADT operators, as the checks occupy a lot of memory.

Note:

This flag should be used for testing purposes but uses lots of code size within the target.

Initial setting	set

- XERANGE
  XTESTF

These flags define that an SDL syntype is to be checked against its defined range. If the flags are not set, the code to perform this check is left out. The flag should be set for allowing better error checks, but may be unset for optimizing the target program code, as the checks occupy a lot of memory.

Initial setting	set

Size of Variables

- X_COMPACT_BOOL

  This flag defines that an SDL boolean is translated to unsigned char, when it is set. Otherwise each SDL boolean is translated to integer. Usually this flag should be left set, as it is in the default setting. An exception might become true when a C compiler can handle integer values more efficient than char values.

Initial setting	set

- X_SHORT_REAL

  When this flag is defined, then each SDL real value is translated to a float in C. If it is not defined, Cmicro assumes SDL real values are C double values. The user must take care that at no place in the SDL system there is a real value used that exceeds the maximum range of the appropriate C type.

Initial setting	set

- X_LONG_INT

  By defining this switch, the SDL predefined sorts, integer and natural are translated to long, otherwise these are translated to integer. The user must take care that at no place in the SDL system there is a real value used that exceeds the maximum range of the appropriate C type.

Initial setting	unset

Use of Memory

Memory Management

- XMK_USE_SDL_MEM

  If this flag is defined, then the Cmicro Kernel can use the dynamic memory management functions contained in the `ml_mem` module (`xmk_Malloc()`, `xmk_Calloc()` and `xmk_Free()`). This is necessary if the compiler in use has no `malloc/free` or if the user wishes to modify the standard behavior of these functions, i.e. using a best fit, instead of first fit searching algorithm. The user should refer to "Dynamic Memory Allocation Functions – Cmicro" on page 3544 also.

Initial setting	unset
Reset when unset	XMK_USE_memshrink

- XMK_USE_MIN_BLKSIZE

  By setting this define it is possible to organize the dynamic memory allocation in a similar way as arrays in C. If all the allocated blocks in the memory occupy the same space then no fragmentation problems occur.

Initial setting	unset

- XMK_USE_memshrink

  If this flag is defined, then the `xmk_Memshrink()` function of the `ml_mem` module can be used. This function delivers the opportunity to free the unused space of a memory block that was requested previously with `xmk_Malloc()` or `xmk_Calloc()`.

Initial setting	unset
Depend on	XMK_USE_SDL_MEM

- XMK_USE_memset

  If this flag is defined, then the `memset()` function of the `ml_mem` module is compiled.

Initial setting	unset

- XMK_USE_memcpy

  If this flag is defined, then the `memcpy()` function of the `ml_mem` module is compiled.

Initial setting	unset

- XMK_CPU_WORD_SIZE

  For allocating memory the pointer to the end of the allocated memory must be dividable by XMK_CPU_WORD_SIZE. This value is set to 8 as default, but should be decreased to 4, 2 or even 1 if the CPU's layout allows it.

Initial value	8
Depend on	XMK_USE_SDL_MEM

- XMK_MEM_MIN_BLKSIZE

  The requested block size is rounded to a value of XMK_MEM_MIN_BLKSIZE, if the XMK_MEM_MIN_BLKSIZE is greater than the requested block size.

Initial value	64
Depend on	XMK_USE_MIN_BLKSIZE

- XMK_MAX_MALLOC_SIZE

  If the Cmicro memory functions (`xmk_Malloc()`, `xmk_Calloc()` and `xmk_Free()`) are used, the buffer that is used for dynamic memory allocation is to be initialized with `xmk_MemInit()`. The size of the buffer can be given here.

Initial value	1024
Depend on	XMK_USE_SDL_MEM

String Functions

- XMK_USE_strcpy

  If this flag is defined, then the `strcpy()` function of the `ml_mem` module is compiled.

Initial setting	unset

- XMK_USE_strncpy

  If this flag is defined, then the `strncpy()` function of the `ml_mem` module is compiled.

Initial setting	unset

- XMK_USE_strcmp

  If this flag is defined, then the `strcmp()` function of the `ml_mem` module is compiled.

Initial setting	unset

- XMK_USE_strlen

  If this flag is defined, then the `strlen()` function of the `ml_mem` module is compiled.

Initial setting	unset

SDL environment

- XMK_USE_xInitEnv

  This flag enables the C function xInitEnv() which is generated as a template in file env.c by the Targeting Expert.

Initial setting	set

- XMK_USE_xInEnv

  This flag enables the C function xInEnv() which is generated as a template in file env.c by the Targeting Expert.

Initial setting	set

- XMK_USE_xOutEnv

  This flag enables the C function xOutEnv() which is generated as a template in file env.c by the Targeting Expert.

Initial setting	set

- XMK_USE_xCloseEnv

  This flag enables the C function which is generated as a template in file env.c by the Targeting Expert.

Initial setting	set

- XMK_USE_SEND_ENV_FUNCTION

  This flag make it possible to use an alternative function to send signals to the environment. This is needed if the used compiler does not support re-entrant functions. The disadvantage of this implementation exists in the missing error checks. See "Alternative Function for sending to the Environment" on page 3595

Initial setting	unset

- XMK_ADD_STDIO

  The Cmicro Package is mainly intended for target implementation. However on some occasions the user may wish to use OS features. Normally this flag is undefined when compiling for the target system but may be defined if `stdio` is available and required on the target system.
  Defining `XMK_ADD_STDIO` without setting `XMK_ADD_PRINTF` allows the user to exclude the default `printf`s of the Cmicro Library and instead implement user defined `printf`s.

Initial setting	unset
Reset when unset	XMK_ADD_PRINTF

Automatic Scaling Included in Cmicro

The flags described in this section are automatically generated into the file `sdl_cfg.h` by the Cmicro SDL to C Compiler. The purpose is to exclude parts of the C code in order to reduce the generated code. Features or functions which are not required in the SDL description produce no (or only a small) overhead in the generated code.

If the user does not wish to employ automatic scaling facilities, for example if test and debugging proves too difficult, then simply define the flags XMK_USE_NO_AUTO_SCALING.

All the flags discussed in this section are used on the whole SDL system i.e. it is not possible to define flags for processes separately.

The Targeting Expert will use these flags, too, to optimize and ease the manual scaling.

- XMK_USED_ONLY_X_1

  If this flag is generated as defined, then there is no process defined in the system, where N is > 1 in the process declaration (x, N). This allows memory to be saved as the overhead for process addressing in the system is reduced. The need for numbering of process instances is also eliminated.

- XMK_USED_DYNAMIC_CREATE

If this flag is generated as defined, the SDL system uses the dynamically created process feature of SDL. (It only requires one create-symbol in the SDL system to enable this flag to be set). This of course constitutes a bigger Cmicro Kernel.

- XMK_USED_DYNAMIC_STOP

  If this flag is generated as defined, the SDL system uses the dynamic process stop feature of SDL. (It only requires one stop-symbol in the SDL system to enable this flag to be set). This of course constitutes a bigger Cmicro Kernel.

- XMK_USED_SAVE

  If this flag is generated as defined, the SDL system uses the save feature of SDL. Using save means that there has to be additional overhead in each signal stored in the SDL queue (each signal is tagged as save or not save).
  The save construct, although a useful construct for manipulation of the SDL FIFO queue mechanism, unfortunately imposes a large code overhead. The user should avoid the utilization of this construct where possible.

- XMK_USED_TIMER

  If there is a minimum of one timer declaration in the system, then this flag is generated as defined. If no timer declarations exist in the SDL system, the timer handling functions are excluded.

Note:

It is possible to implement a user defined timer model by un-defining these flags and defining some other macros.

- XMK_HIGHEST_SIGNAL_NR

  This define gives the amount of signals and timers used in the SDL system in sum. It is used to scale internal buffers of the Cmicro Package.

- XMK_USED_SIGNAL_WITH_PARAMS

  If no signals with parameters are defined in the SDL system, then this flag is generated as defined which reduces the overall code size.

Note:

It is not recommended in each case (because it is not SDL conform), but is possible to send parameters via global parameters by using C code in SDL.

- XMK_USED_TIMER_WITH_PARAMS

  This define is generated if there is at least on timer with parameter used in the system. If this is the case, additional functionality for this construct is added when the kernel is compiled. The timer operations will result in producing more overhead, both symbol by symbol and overhead in the kernel compared with the case if there are no timers with parameters defined.

- XMK_USED_SENDER, XMK_USED_PARENT, XMK_USED_OFFSPRING, XMK_USED_SELF

  The above flags are generated as defined if the user uses the appropriate addressing construct within SDL, that is sender, parent, offspring or self.
  If “to” in output actions is not used, none of the above flags is generated as defined.
  The define `XMK_USE_PID_ADDRESSING` depends on the above flags.

- XMK_USED_PWOS

  If procedures without states are contained in the SDL system, this flag is defined. Procedures with states are not supported by the Cmicro Package.

Automatic Dimensioning in Cmicro

Some resources of the Cmicro Library are automatically dimensioned. These are described in the following subsections.

- MAX_SDL_PROCESS_TYPES

  The Cmicro SDL to C Compiler counts the amount of process types in the system and generates this define. This is used to dimension some tables used in the generated code, the Cmicro Kernel and the SDL Target Tester.

- MAX_SDL_TIMER_TYPES

The Cmicro SDL to C Compiler counts the amount of timer types in the system and generates this define. This is used to dimension some tables used in the generated code, the Cmicro Kernel and the SDL Target Tester.

- MAX_SDL_TIMER_INSTS

  The Cmicro SDL to C Compiler counts the amount of instances of timers in the system and generates this define. This is used to dimension some tables used in the generated code, the Cmicro Kernel and the SDL Target Tester.

Adaptation to Compilers

In this section the steps are explained that must be carried out in order to deal with a new C compiler which is not yet in the list of available compilers.

The following parts may be modified manually:

- In `mk_stim.c`, there are template functions as templates which have to be modified to adapt another hardware/compiler to the SDL system time (Now).
- In `mk_cpu.c`, there are templates for functions which represent hardware access. Those are required if the preemptive Cmicro Kernel is used.

The following parts can be added by using the Targeting Expert:

- A compiler specific header file `user_cc.h` which will automatically be included in `ml_typ.h`. Please view "Compiler Definition for Compilation" on page 2910 in chapter 59, *The Targeting Expert*.
- An entry in the Targeting Expert's configuration files. Please view "Compiler Definition for Compilation" on page 2910 in chapter 59, *The Targeting Expert*.

List of Available C Compilers in ml_typ.h

C compilers are to be selected by the user by choosing from an available list with the help of the Targeting Expert.

The Targeting Expert will generate a C define into the `ml_mcf.h` file. When compiling Cmicro sources, the right C compiler section in `ml_typ.h` is selected by using a `#ifdef <compilername>` construct.

The following `<compilername>` defines are currently defined in `ml_typ.h`:

Compilation switch	Meaning
GCC	The GNU C++ compiler for workstations
GNU80166	The GNU UNIX C compiler for Siemens 80C166 microcontrollers
TCC80166	The BSO/Tasking DOS C compiler for Siemens 80C166 microcontrollers
TCC80C196	The BSO/Tasking DOS C compiler for INTEL 80196 microcontrollers
IARC51	The Archimedes/IAR UNIX C compiler for INTEL 8051 microcontrollers
IARC6301	The Archimedes/IAR DOS C compiler for Hitachi 6301 microcontrollers
KEIL_C51	The Franklin/Keil DOS C compiler for INTEL 8051 microcontrollers
KEIL_C166	The Franklin/Keil DOS C compiler for Siemens 80166 microcontrollers
TMS320 MSP58C80	The Texas Instruments DOS C compiler for TMS 320C2x/C5x microcontrollers
IARC7700	The Archimedes/IAR DOS C compiler for Melps 7700 microcontrollers
HYPERSTONE	Hyperstone 5.07 C compiler with HyRTK real-time kernel
MCC68K	The MicroTech DOS C compiler for Motorola 68k microprocessors
MICROSOFT_C	The Microsoft C++ compiler
ARM_THUMB	The Thumb compiler for ARM microcontrollers
ICC_HC12	The ICC12 5.0 Compiler for HC12 microcontrollers

If none of these compiler flags is defined during compilation the file `user_cc.h` (generated by the Targeting Expert) will be included in `ml_typ.h`.

The remaining part of the code of the Cmicro Library should be compilable without performing any modifications. If there are problems with adapting a new compiler or hardware please contact Cmicro technical support.

The user may however decide to define his own C compiler. This is explained in the following subsection.

Introducing a new C Compiler

Adding a new C Compiler to the Project

The first alternative to add a new user defined C compiler, is to add that compiler to a user's project. This can be achieved with the Targeting Expert.

- Give the C compiler a name for using it within Cmicro. The name should as a recommendation be in the notation of C macros, that is, it should be 8 to approximately 16 characters long in uppercase letters.
- Below the Targeting Expert's *Edit* menu there is a choice called "Add a new C Compiler" by which the new C compiler can be defined.
- It is also possible to remove the user defined C compiler later on by using this menu.

These changes are not stored within the installation, but within the user's project directory (which is the target directory that was chosen when the Targeting Expert was started).

Do the C Compiler Adaptations

- Create a new file with the name `user_cc.h`. If a C compiler is selected, which is not in the list of available C compilers in the Cmicro product, the `user_cc.h` is included automatically. This can easily be done by using the Targeting Expert's *Edit* menu "Edit compiler section".
- Take care that the right path names are specified when the C compiler is invoked. There usually should be something like a "`-I.`" option which must point to the path in which `user_cc.h` is stored.

Description of user_cc.h

The things that are to be defined by the user are:

- Is the compiler able to handle function prototypes, as defined in ANSI-C?

Yes	`#undef  XNOPROTO` `#define XPP(x)     x` `#define PROTO(x)   x`
No	`#define XNOPROTO` `#define XPP(x)` `#define PROTO(x)`

- Is the controller faster in accessing character values or integer values?

char	`#define xmk_OPT_INT char`
integer	`#define xmk_OPT_INT integer`

- Does the compiler support variables stored in registers?

Yes	`#undef  X_REGISTER` `#define X_REGISTER >register<`
No	`/* Nothing to do for X_REGISTER */`

`>register<` is the compiler specific command to store variables in registers.

- Default setting:
  `#define xptrint unsigned long`
  This setting should work for the very most compilers In a few cases it is necessary to define `xptrint` as `unsigned int`.

- Default setting:
  `#define xint32 long`
  This setting should work for the very most compilers. In a few cases it is necessary to define `xint32` as `int`.

- Is the compiler able to handle the C keyword `const` in the correct way?

Yes	`#define XCONST const`

No	`#undef XCONST`

In general, the compilers which are able to produce ROM-able code, can handle the keyword `const`. Compilers may generate false object code, if using `const`.

- Is it a UNIX system for which code is to be compiled for?

Yes	`#define XMK_UNIX`
No	

- Is it an MS Windows system for which code is to be compiled for?

Yes	`#define XMK_WINDOWS`
No	

- Critical paths must be enabled and disabled:

```
#undef  XMK_END_CRITICAL_PATH
#define XMK_END_CRITICAL_PATH  \
               if (xmk_InterruptsDisabled) \
               {\
                 xmk_InterruptsDisabled--;\
                 if (!xmk_InterruptsDisabled) \
                 { ENABLE; }}

#undef  XMK_BEGIN_CRITICAL_PATH
#define XMK_BEGIN_CRITICAL_PATH \
                 DISABLE;\
                 xmk_InterruptsDisabled++;
```

`ENABLE` and `DISABLE` are the compiler specific command to allow/prevent interrupts and must be filled.

- A #include of <stdio.h>, if supported by the compiler (not all the target compilers do). Write:

```
#ifdef XMK_ADD_STDIO
  #include <stdio.h>
#endif
```

- Which include header files must be added?

  For example, for the IARC51:

```
#define "io51.h"
```

is used. For the GNU80166:

```
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <c166.h>
```

is used.

- The include header files containing the prototypes for dynamic memory allocation and string and memory functions must be added. This subsection should look like this for allowing the user to select either the memory functions from the C compiler or from Cmicro:

```
#ifdef XMK_USE_SDL_MEM
  #include "ml_mem.h"
#else
  #include "string.h"
#endif /* XMK_USE_SDL_MEM */
```

It must be checked if `malloc(), free(), calloc()` etc. are really prototyped in `string.h`, on other compilers it might be `stdlib.h`. Cmicro always calls `xAlloc()` and `xFree()` which are given as examples in the `mk_cpu.c` file below the template directory. For more information on dynamic memory allocation please view the subsection "Dynamic Memory Allocation" on page 3543.

Defining the SDL System Time Functions in mk_stim.c

The following functions exist in the module `mk_stim.c`:

- `void xmk_InitSystime(void)`
  Initialize the hardware registers to support the system time
- `void xmk_DeinitSystime(void)`
  Give up to use the system time. This normally cannot happen, but in some applications, where the SDL system is stopped and restarted again during run-time, it may prove useful.
- `void xmk_SetTime(xmk_T_TIME)`
  This function sets the system-time to the given value. Called by Cmicro Kernel in the case when an overrun in the time value is detected.

- `xmk_T_TIME xmk_NOW(void)`
  This is the most important C function to handle SDL system-time. This function returns the absolute time, which is by default defined as a `long` value (`xmk_T_TIME`).

Usually, the above functions are conditionally compiled. To make the functions available in the target system, at least one timer in SDL must be declared. The functions are not included if there is no timer declared but duration, time or now is used in SDL. This will lead to compilation errors.

To make timers in SDL operable, absolute time must be implemented. This can be reached by using a hardware free running counter or by using a timer interrupt service routine, which clocks a global variable containing the absolute time.

Bare Integration

This section deals with functions that must be adapted by the user in order to connect the SDL system to the environment i.e. connection to target hardware, and the environment.

Implementation of Main Function

The user may decide to implement his own `main()` function body when it comes to target application in a bare integration. A default `main()` function is included in the template file `mk_user.c`.

The implementation of a first `main` function looks like this:

Example 606 ----------

```
main ()
{
  xmk_InitQueue();
  xmk_InitSDL ();
  xmk_RunSDL ();
}
```

Of course this example does nothing in order to start the SDL Target Tester. The only possible way to get a very simple trace output is to define:

```
#define XMK_ADD_PRINTF
```

in `ml_mcf.h` with the help of the Targeting Expert. This will include calls to the C function `xmk_printf()` at several places in the generated C code and the Cmicro Library. The `xmk_printf()` function is available as an example in the file `mk_cpu.c`.

The user can use the `main()` functions delivered with the Cmicro Kernel to have full access to all Cmicro features. This delivered `main()` function is a template and can be modified to add or remove functionality.

Integrating Hardware Drivers, Functions and Interrupts

There is no extra handling for hardware drivers, hardware functions and interrupt service routines. It is necessary to implement an interface in the user's application. Hardware drivers, hardware functions and interrupt service routines are seen from the SDL system as the environment. The user has to implement the interface between the SDL system and the environment as it is described in the following subsections.

Critical Paths in the Cmicro Library

As with every real time application the Cmicro Library has to deal with critical paths.

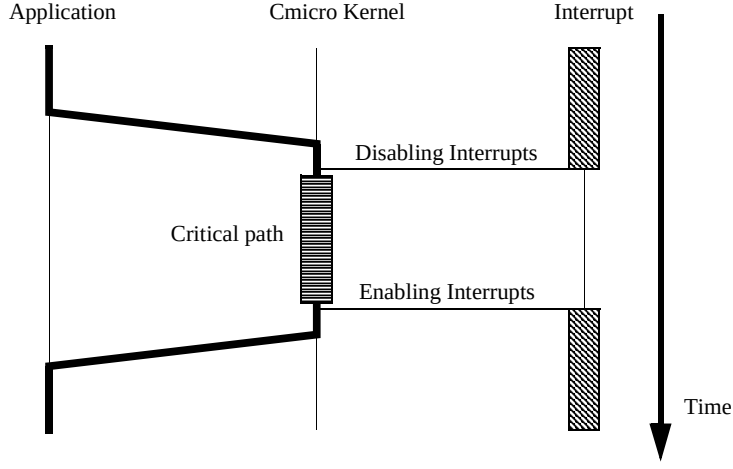


Figure 598: Critical paths

These are well defined in the source code using two macros namely `XMK_BEGIN_CRITICAL_PATH` and `XMK_END_CRITICAL_PATH` (see file `ml_typ.h`/compilersections).

The macros are expanded to compiler specific calls which disable/enables interrupts. The calls are counted using an internal variable, thus after having called `XMK_BEGIN_CRITICAL_PATH` n times `XMK_END_CRITICAL_PATH` has also to be called n times, to enable the interrupts again.

In the Cmicro Library these macros are used to disable/enable interrupts only. The application should take care not to handle interrupts without using these macros or at least evaluating the internal counter.

Caution!

When it comes to implementing the interrupt handling routines it is of the greatest importance to know about the Cmicro Library's handling of critical paths. Unpredictable results may result from a careless implementation!

Initializing the Environment / Interface to the Environment

xInitEnv()

There is one C function called `xInitEnv()` available as a template in the generated file `env.c`. The user should fill this module out appropriately, thus implementing connection to the environment (drivers, interrupt service routines and so on).

Receiving Signals from the Environment

There are two possibilities to send signals into the SDL system. In all cases, the user has to specify an SDL input symbol in the process which has to consume the signal. The possibilities are:

1. Polling external events.
   This is done by the C function `xInEnv()`, which is called by the Cmicro Kernel after each transition.

2. Directly sending signals into the SDL system.
   For example directly in an interrupt service routine. This is possible by using the `XMK_SEND_ENV()` function. This function directly operates on the SDL queue. If interrupt service routine is used and can result in signal sending the user need to manually define flag XMK_USE_INTERRUPT_SERVICE_ROUTINE to enable additional needed critical paths in kernel.

Caution!

For each signal sent into the system, the user must ensure that the addressed receiver process instance exists. Each pid value is checked for consistency by the Cmicro Kernel. In the case of an inconsistency the `ErrorHandler()` is called with an error message.

xInEnv()

Parameters:

```
In/Out: -no-
Return: -no-
```

In the case when no interrupt service routine is available to handle an external event, the Cmicro Kernel can poll external events. The user must use the C function `xInEnv()` in the file `env.c` generated by the Targeting Expert.

For example, a hardware register can then be checked in order to establish if its value has changed since the last call.

If an external signal is detected, then one of the `xmk_Send*()` functions is to be called thus enabling the signal to be put into the SDL queue.

Note:

There is a file called `<systemname>.ifc`, which is generated by the Cmicro SDL to C Compiler. It is necessary to include this file together with `ml_typ.h` before defining `xInEnv()`. This is necessary to have access to all the objects that are generated by the Cmicro SDL to C Compiler. The user should also make sure that this file is generated, because the analyzer's make menu contains an option for this.

Example 607: `xInEnv()` ----------

Assume, a hardware register where the value 0 is the start-value and where any other value means: data received. Two signals, namely `REGISTER_TO_HIGH`, and `REGISTER_TO_LOW` are to be defined in the SDL system, with a process receiving these. Then the user supplied C code should look like this (the `<p>` below stands for the automatically generated prefix, see the file `sdl_cfg.h`):

```
void xInEnv (void)
{
  XMK_SEND_TMP_VARS
  static state = 0;
              /* state, to detect changes in the */
              /* hardwareregister               */

 /* BEGIN User Code */
 if ((state==0) && (hw_register != 0))
 /*   END User Code */
 {
   XMK_SEND_ENV (REGISTER_TO_HIGH,
                 xDefaultPrioSignal,
                 0,
                 NULL,
 /* BEGIN User Code */
   GLOBALPID(XPTID_<p>_ReceiverName,0));
   state = 1;
 /*   END User Code */
   return;
 }

 /* BEGIN User Code */
 if ((state==1) && (hw_register == 0))
 /*   END User Code */
 {
   XMK_SEND_ENV (REGISTER_TO_LOW,
                 xDefaultPrioSignal,
                 0,
                 NULL,
/* BEGIN User Code */
   GLOBALPID(XPTID_<p>_ReceiverName,0));
   state = 0;
/*   END User Code */
    return;
  }

  return;
} /* END OF SAMPLE */
```

Caution!

The function `xInEnv()` contained in the file `env.c` is generated by the Targeting Expert. To make sure that changes done by the user will not be lost when generating again it is allowed to modify this file only between the comments `/* BEGIN User Code */` and `/* END User Code */`.

Furthermore it is not allowed to remove these comments or to add similar at other places which could especially happen when copy and paste is used during editing.

XMK_SEND_ENV()

Parameters:

```
In/Out: xPID Env_ID

        xmk_T_SIGNAL sig

        #ifdef XMK_USE_SIGNAL_PRIORITIES
          xmk_T_PRIO prio
        #endif

        #ifdef XMK_USED_SIGNAL_WITH_PARAMS
          xmk_T_MESS_LENGTH data_len,
          void xmk_RAM_ptr  p_data
        #endif

        #ifdef XMK_USE_RECEIVER_PID_IN_SIGNAL
          xPID Receiver
        #endif

Return: -no-
```

It is possible to send signals directly into the SDL system by calling the C function `XMK_SEND_ENV()`, no matter whether preemption is used or not. No conflict occurs, if an interrupt service routine uses a `XMK_SEND_ENV()` function parallel to the C function `xInEnv()`.

This function is to be called when a signal is to be sent into the SDL system, e.g. within the users `xOutEnv` function. It must be called with one more parameter than the `xmk_Send` function, which is the first parameter `Env_ID`. This parameter must be set to `ENV` by the user.

The macro internally uses some variables which are to be declared before the macro can be used. For example in the xOutEnv function the `XMK_SEND_TMP_VARS` macro must be introduced for declaring these variables.

The function is implemented as a macro in C.

Use the template in the previous subsection to see details how to use the `XMK_SEND_ENV()` functions.

Sending Signals to the Environment

There are several possibilities to send signals to the environment:

1. Using simple SDL output.
   The Cmicro Kernel is involved in the output operation.

2. Using the `#EXTSIG` directive in the SDL output.
   The user can define his own output operation, like writing to a register in a single in-line-assembler command.

3. Using the `#ALT` directive in the SDL output.
   A variant of alternative 2.

Alternative 1 is the most SDL like alternative because it does not use non SDL constructs, like directives. This alternative should be selected, where possible because it makes the diagrams more SDL like and MSCs more readable.

Alternative 2 and 3 are probably the alternatives with higher performance.

Alternative 1 is described below. Alternative 2 and 3 are already well described in chapter 65, *The Cmicro SDL to C Compiler*.

xOutEnv()

Parameters:

```
   In/Out:
            xmk_T_SIGNAL         xmk_TmpSignalID

            #ifdef XMK_USE_SIGNAL_PRIORITIES
            xmk_T_PRIO           xmk_TmpPrio
            #endif

            #ifdef XMK_USED_SIGNAL_WITH_PARAMS
              xmk_T_MESS_LENGTH xmk_TmpDataLength,
              void xmk_RAM_ptr  xmk_TmpDataPtr
            #endif

            #ifdef XMK_USE_RECEIVER_PID_IN_SIGNAL
              xPID xmk_TmpReceiverPID
            #endif

   Return: xmk_OPT_INT
```

The function `xOutEnv()` exists as a template in the generated module `env.c`.

Each time an SDL output occurs, the generated C code calls the C function xmk_Send or xmk_SendSimple. After performing some checks, and if the signal is to be sent to the environment, the C function xOutEnv() is called with all the parameters necessary to represent the signal. The parameters are explained in the following.

With the parameter xmk_TmpSignalID, the signal ID is to be specified.

With the xmk_TmpPrio parameter, the signal's priority is to be specified (if conditionally compiled). If the XMK_USE_SIGNAL_PRIORITIES is not defined, the signal priorities which are specified with #PRIO in the diagrams is just ignored. The use of signal priorities is not recommended because this violates SDL. A few bytes can be spared if signal priority is not used. See also XMK_USE_SIGNAL_PRIORITIES.

With the parameter xmk_TmpDataLength, the number of bytes of signal parameters is to be specified. The number of bytes is evaluated by using a sizeof (C struct) construct. If the signal carries no parameters, this value is set to 0.

With the xmk_TmpDataPtr parameter, a pointer to the memory area containing the parameter bytes of the signal is given. The memory area is not treated as dynamically allocated within this function. Because the function copies the parameter bytes, the caller may use any temporary memory (for example memory allocated from the C stack by declaring a C variable). This parameter should be set to NULL if no parameter bytes are to be transferred (if conditionally compiled).

The parameters xmk_TmpDataLength and xmk_TmpDataPtr are compiled conditionally. The XMK_USED_SIGNAL_WITH_PARAMS is automatically generated into the sdl_cfg.h file, from the Cmicro SDL to C Compiler. For tiny systems, if there are no SDL signals with parameters specified, this is undefined. It will reduce the amount of information which is to be transferred for each signal, with a few bytes. See also XMK_USED_SIGNAL_WITH_PARAMS.

With the last parameter xmk_TmpReceiverPID, the PID of the receiving process is to be specified (if conditionally compiled). The parameters xmk_TmpDataLength and xmk_TmpDataPtr are compiled conditionally, see also XMK_USE_RECEIVER_PID_IN_SIGNAL.

If XMK_USE_RECEIVER_PID_IN_SIGNAL is not defined, the user must implement the C function xRouteSignal which is responsible to derive the receiver from the signal ID in that case. Using xRouteSignal

is recommended only if the last few bytes must be spared for transferring of signals.

> **Note:**
>
> There is a file called `<systemname>.ifc`, which is generated by the Cmicro SDL to C Compiler. It is necessary to include this file together with `ml_typ.h` before defining `xOutEnv()`. This is necessary to have access to all the objects that are generated by the Cmicro SDL to C Compiler. The user should also make sure that this file is generated, because the analyzer's make menu contains an option for this.

Evaluate the Signals's Sender

The sender of the signal can be retrieved by the global variable `xRunPID` of type `xPID`. Users should remember, that in a system with only (x,1) process declarations, this pid represents the process type of the sender. In a system in which multiple process instances of the same process type appear, this pid represents the process type plus the process instance of the sender. Probably it is required to make decisions depending on the sender of the signal. The user can evaluate the process type of the sender by using the C expression `EPIDTYPE(xRunPID)`, i.e.:

Example 608: Evaluating the Process Type ––––––––––––––––––––

```
unsigned char ptype;
ptype = EPIDTYPE(xRunPID);
```

––

Remember also, if it is required to signal to a specific pid in the environment, this requires dynamic signalling where a process in the environment establishes communication to a process in the system or the other way around.

Checking the Signal's Receiver

Under normal circumstances it is not necessary to check the receiver in `xOutEnv()` as the Cmicro Kernel calls `xOutEnv()` only, if the environment is recognized as the signal's receiver.

Caution!

The users have to ensure that signals to the environment are consumed in the environment by returning `XMK_TRUE` in `xOutEnv()`.

Signal Parameters

If the signal to be sent to the environment contains parameters, it is necessary to copy these parameters to the data area of the environment as the signal and its parameters are deleted after signal consumption.

For each signal carrying parameters, there is a typedef struct generated into the `<systemname>.ifc` file. Please view Example 609.

Return Values of xOutEnv()

The user must ensure that signals which are to be sent to the environment are consumed by the environment. In `xOutEnv()` this is done by returning the values `XMK_TRUE` or `XMK_FALSE`.

- `XMK_TRUE`
  The signal has been consumed by the environment.
- `XMK_FALSE`
  The signal is NOT consumed by the environment.

An Easy Example

Assume, for example, a process which has to send a signal with parameters to the environment. The code the user has to write into the `xOutEnv()`-function, looks as follows:

Example 609: `xOutEnv()`

```
xmk_OPT_INT xOutEnv (xmk_T_SIGNAL      xmk_TmpSignalID,
                     xmk_T_PRIO        xmk_TmpPrio,
                     xmk_T_MESS_LENGTH xmk_TmpDataLength,
                     void *            xmk_TmpDataPtr,
                     xPID              xmk_TmpReceiverPID )

{
  xmk_OPT_INT xmk_TmpResult = XMK_FALSE;

  switch (xmk_TmpSignalID)
  {
    case SDL_Signal1 :
      {
        /* BEGIN User Code */
        int temp;
        temp = (yPDef_z4_SDL_Signal1*)xmk_TmpDataPtr->Param1
        UserFunction(temp);
        /*   END User Code */
        xmk_TmpResult = XMK_TRUE; /* signal is */
                                  /* consumed  */
      }
      break ;

   case SDL_Signal2 :
      {
        /* BEGIN User Code */
        char g;
        int  k;
        g = (yPDef_z5_SDL_Signal2*)xmk_TmpDataPtr->Param1;
        k = (yPDef_z5_SDL_Signal2*)xmk_TmpDataPtr->Param2;
        UserFunction2( g, k );
        /*   END User Code */
        xmk_TmpResult = XMK_TRUE; /* signal is */
                                  /* consumed  */
      }
      break ;

   default :
        xmk_TmpResult = XMK_FALSE;/* signal is NOT */
                                  /* consumed   */
                                  /* and to be handled */
                                  /* by the Cmicro Kernel */
        break ;
  }
  return( xmk_TmpResult );
}
```

Caution!

The function `xInEnv()` contained in the file `env.c` is generated by the Targeting Expert. To secure that changes done by the user will not be lost when generating again it is allowed only to modify this file between the comments `/* BEGIN User Code */` and `/* END User Code */`.
Furthermore it is not allowed to remove these comments or to add similar ones at other places.

Inter-Processor-Communication

> **Note:**
>
> If inter-processor-communication is to be performed, the user has to define a unique protocol between the communicating processors. In small applications with a restricted range, it might be enough to copy C structures, but usually this causes problems when it comes to redesigning the hardware.

Closing the Environment / the Interface to the Environment

xCloseEnv()

Parameters:

```
In/Out: -no-
Return: -no-
```

There is one C function called `xCloseEnv`() available as a template in the generated C module `env.c`.

The user should fill this function out appropriately thus realizing disconnection from the environment (drivers, interrupt service routines and so on). Disconnection can make sense if a re-initialization or a software reset is to be implemented.

The `SDL_Halt` function is mapped to `xCloseEnv()` in Cmicro.

SDL System Time Implementation

Included in the Cmicro Kernel, there are some template functions for the implementation of SDL system time. All these functions are contained in the module `mk_stim`. The functions `xmk_NOW()` or `xmk_SetTime()` must be implemented newly if a new method for accessing the hardware system time is to be implemented. In many cases, the standard C library function `time()` is available, which is used in most of the C compiler adaptations that are already been made.

It is necessary though to implement a function which makes the variable `SystemTime` topical. This will be an interrupt service routine in most cases.

Please view "Defining the SDL System Time Functions in mk_stim.c" on page 3528.

Getting the Receiver of a Signal – Using xRouteSignal

Parameters:

```
In/Out: xmk_T_SIGNAL sig
Return: xPID
```

The user can remove the receiver's xPID from the signal structure by resetting the flag "XMK_USE_RECEIVER_PID_IN_SIGNAL" on page 3491. In this case, the function `xRouteSignal()` has to be filled as the Cmicro Kernel needs to know which process likes to receive the current signal.

Example 610: Function `xRouteSignal()`

```
/*
** the <p> below stands for the automatically generated prefix
*/

xPID xRouteSignal  ( xmk_T_SIGNAL sig )
{
   /*
   ** Please insert appropriate code here to map
   ** Signal ID's to Process - Type - ID's
   ** (XPTID_ProcessName in generated code).
   ** Keep in mind that this function might be
   ** called from within a critical path.
   ** Include <systemname>.ifc to get signal names
   ** and process' XPTID
   */
   switch (sig)
   {
    /*
    **  S D L   T i m e r s ...
    */
    case SDL_Timer1: return (XPTID_<p>_ProcessName_A)
                     break;
    case SDL_Timer2: return (XPTID_<p>_ProcessName_B)
                     break;
    case SDL_Timer3: return (XPTID_<p>_ProcessName_A)
                     break;
    case SDL_TimerN: return (XPTID_<p>_ProcessName_C)
                     break;

    /*
    **  O r d i n a r y  S D L  S i g n a l s ...
    */
    case SDL_Signal1: return (XPTID_<p>_ProcessName_B)
                      break;
    case SDL_Signal2: return (XPTID_<p>_ProcessName_A)
                      break;
    case SDL_Signal3: return (XPTID_<p>_ProcessName_A)
                      break;
             ......

    case SDL_SignalN : return (XPTID_<p>_ProcessName_C)
                      break;

    default: ErrorHandler (ERR_N_NO_RCV);
             return (xNULLPID)
             break;
   }
}
```

Dynamic Memory Allocation

General

Dynamic memory allocation in real life always introduces problems, which are:

- The memory occupation cannot be evaluated by the user, so that it is impossible to configure the dynamic memory. If all objects are allocated statically and if the memory occupation exceeds the available memory, this results in errors during compilation/linking, or at least a memory map file can be viewed.
- If dynamic memory allocation is used, then, after the program has executed for a time, usually memory leaks are the result. Memory leaks are fatal because there might be enough memory, but it is even impossible to allocate one more block.
- If there is no more free dynamic memory available, then it is up to the user to decide how to continue in the program. In any case, the program should be terminated, started again or any similar reaction is to be programmed.

Normally, Cmicro tries to prevent any use of dynamic memory allocation, but there are the following exceptions, in which this is impossible:

- When a signal with too many parameters is to be sent to another process. See "Signals, Timers and Start-Up Signals" on page 3460.
- When an SDL sort is used requiring dynamic memory management, like the charstring sort.
- If the SDL Target Tester is used, because there is dynamic memory allocation in the start-up phase and for each transmit buffer.

There are at least two possibilities to implement dynamic memory allocation namely:

- The dynamic memory management from the C Compiler or operating system can be used.
- The dynamic memory management from Cmicro can be used.

When it comes to targeting, the user decides upon the dynamic memory allocation manager. In the following, the both possibilities are explained.

Dynamic Memory Allocation Functions – Compiler or Operating System

If the C compiler, that the user is using in the target environment, provides dynamic memory allocation functions, it is possible to use these functions. A few steps must be carried out to include the dynamic memory allocation functions of the C compiler or operating system, which are:

- The user must introduce his own C compiler section, please refer to the subsection "Introducing a new C Compiler" on page 3525.
- In the new C compiler section, the correct header files must be introduced with `#include`. The user should refer to the manuals of the C compiler or operating system.
- The SDL Target Tester, Cmicro Library and Cmicro Kernel always allocate dynamic memory by using the functions in the file `mk_cpu.c`.
- The file `mk_cpu.c` below the Cmicro template directory must be copied into the user's project directory. Any modifications should be performed on the private copy of `mk_cpu.c`.
- The `mk_cpu.c` file contains two C functions called `xAlloc()` and `xFree()`. Within these functions, the user should call the appropriate dynamic memory allocation functions of the C compiler or operating system.

Dynamic Memory Allocation Functions – Cmicro

Below the Cmicro kernel directory there is a C module `ml_mem.c` that implements a dynamic memory allocator. The C module `ml_mem.c` can be used for managing memory dynamically for SDL systems, but it is not forbidden to use this module within handwritten C code also.

The module cannot be used if partitioning is to be used. In that case compilation errors will occur. Please refer to "Dynamic Memory Allocation" on page 3543 for an explanation which parts in SDL are to be dynamically allocated. The module provides C functions for initialization, allocation, de-allocation, and getting some information about the current memory status.

There is only one memory pool. The memory pool is to be declared by the user and initialized with the C function `xmk_MemInit` before the

memory pool can be used. Allocations may be performed by calling `xmk_Malloc()` or `xmk_Calloc()`. An allocated block is de-allocated again by calling `xmk_Free()`. So far, the principle behavior is the same as the usual `malloc()`, `calloc()` and `free()` functions from compilers. But the memory pool can probably be cleaned with a Cmicro specific function. There are additional functions which can be used to query the amount of free or occupied space.

Before the memory management functions of Cmicro can be used, a few adaptations are to be made which are explained in the following.

Adaptations That Are to Be Made

Some adaptations are to be made by the user, which are necessary to include the right functions, scale buffers and memory management and take care for general adjusting like alignment of the CPU.

- For general use of the Cmicro memory management functions the user should set the flag `XMK_USE_SDL_MEM` in `ml_mcf.h` with the help of the Targeting Expert. This will make the basic memory allocation functions available. The `ml_mem.c` module must of course be compiled an linked together with the application.
- Adjust the alignment that the CPU is using. There is a macro definition in `ml_mem.c` called `CPU_WORD_SIZE`. The right setting of this macro is basically important for making the dynamic memory allocation functions do work. With this flag, it is possible to define the word size of the target CPU. The value is predefined in `ml_mem.c` but it could be the case that the predefined value is inappropriate for the target system. If the `CPU_WORD_SIZE` value must be redefined, this could be done in `ml_mcf.h` in the user's section. The predefined value for UNIX compilers is 8, the predefined value for `ARM_THUMB` C compiler 4, otherwise a default value of 1 (no alignment) is used.
- Here is a recommendation: The user should use an alignment of 4 (32 Bit) if it is not sure what type of alignment the C compiler / CPU produces. This will work for most cases, but of course probably not if the CPU is 64 Bit CPU. For small systems, this might not be appropriate, because there is an extra overhead of 3 bytes per allocated block. For a CPU like `8051` and derivatives a `CPU_WORD_SIZE` of 1 is appropriate.

- It is possible to introduce a minimum block size per each allocated block. This decreases the risk of getting memory leaks after a while because blocks of the same size can be put together again. If the mechanism of a minimum block size is to be used, then the define XMK_USE_MIN_BLKSIZE is to be used. With the define XMK_MEM_MIN_BLKSIZE the user may define the minimum block size per each allocated block. If the XMK_MEM_MIN_BLKSIZE is not defined from the user, a minimum block size of 64 is predefined.
- The ml_mem.c module contains profiler that keeps track on how many blocks are allocated, how much memory is free and furthermore. The user must define XMK_ADD_PROFILE in order to get the complete functionality. If the SDL Target Tester is used, the flag is automatically predefined.

xmk_CleanPool()
Parameters:
```
In/Out: -no-
Return: size_t
```
This C function returns the amount of occupied memory. It is available only if XMK_USE_SDL_MEM and XMK_SYSTEM_INFO are set.

xmk_GetOccupiedMem()
Parameters:
```
In/Out: -no-
Return: size_t
```
This C function returns the net amount of occupied memory. It is available only if XMK_USE_SDL_MEM and XMK_SYSTEM_INFO are set.

xmk_GetFreeMem()
Parameters:
```
In/Out: -no-
Return: size_t
```
Returns the amount of free memory in sum, which means that the overhead from each block is included. It is available only if XMK_USE_SDL_MEM and XMK_SYSTEM_INFO are set.

xmk_EvaluateExp2Size()
Parameters:
```
In/Out: size_t
Return: size_t
```

This function is either used from Cmicro's dynamic memory allocation functions `xmk_Malloc()` and `xmk_Calloc()` or it may be used directly from the user.

It is available only if `XMK_USE_MIN_BLKSIZE` is set.

The function is declared as:

```
size_t xmk_EvaluateExp2Size ( size_t GivenLength )
```

The function evaluates from the given length a length value, which is in any case a 2 exp N value. This is used to reduce the risk of memory leaks that occur in dynamic memory management systems. It may be used for the memory functions of this module but also for the memory functions of an operating system or C compiler. If the minimum block size is in any case greater than the greatest block that is requested in the target system, then there is no risk for memory leaks. The return result is either the minimum specified with `XMK_MEM_MIN_BLKSIZE`, but might be also one of the following values:

```
64,128,256,512,1024,2048,4096,8192,16384,32768,65536
```

The ?Memory Command

The SDL Target Tester offers a command that allows the user to inspect the status of the dynamic memory. Unfortunately this command can be used only, if the dynamic memory management from Cmicro is used (by setting the `XMK_USE_SDL_MEM` flag in the Targeting Expert).

The command allows the user to check the following:

- Is the dynamic memory correctly initialized after system start-up?
- How is the dynamic memory pool configured?
- At any time during SDL target system execution:
  - what was the greatest block that was allocated?
  - how many blocks are currently in the pool?
  - what is the maximum amount of blocks that have ever been in the pool?
  - physical address of the memory pool (the address of the
  - first block of structure `xmk_T_MBLOCK` (see `ml_mem.c`)

The command presents the following output to the user:

```
M-STATE:Memory pool size incl.overhead     =4096
M-STATE:Current memory fill                =136
M-STATE:Current amount of blocks in pool   =2
M-STATE:Peak hold: Amount of blocks        =29
M-STATE:Peak hold: Largest block           =2048
M-STATE:Overhead per block (in bytes)      =20
```

```
M-STATE:Minimum block size                =0
M-STATE:Memory pool address (hex)         =28478
M-STATE:(Probably there are memory leaks)
```

The command is available only if XMK_USE_SDL_MEM and XMK_ADD_PROFILE are set.

User Defined Actions for System Errors – the ErrorHandler

Parameters:

```
In/Out: int ErrorNo
Return: void
```

Errors, warnings and information can be generated and detected in many places and situations during the lifetime of an SDL system. By fully utilizing the Analyzer, several dynamic error sources can be eliminated at the design stage of development.

Some errors or warnings go undetected by the Analyzer, for example resource errors or real time errors such as memory, performance, or illogical use of SDL.

These errors and warnings are detectable by the Cmicro Kernel and the SDL Target Tester. For a complete list of errors, warnings and information, please view the following subsection "List of Dynamic Errors and Warnings" on page 3550.

As a general rule, for each error and warning there should be an appropriate reaction in the function ErrorHandler() in the mk_user module.

The ErrorHandler() is, as a default implementation, implemented like this:

In the case of a warning message, either:

- The actions defined via the flag XMK_WARN_ACTION_HANG_UP are carried out (see "Reactions on Warnings" on page 3496).
- The actions defined via the flag XMK_WARN_ACTION_PRINTF are carried out.
- The actions defined via the flag XMK_WARN_ACTION_USER and XMK_WARN_USER_FUNCTION are carried out.

In the case of an error message, either:

- The actions defined via the flag XMK_ERR_ACTION_HANG_UP are carried out (see "Reaction on Errors" on page 3498)

- The actions defined via the flag `XMK_ERR_ACTION_PRINTF` are carried out.
- The actions defined via the flag `XMK_ERR_ACTION_USER` and `XMK_ERR_USER_FUNCTION` are carried out.

For more explanations on these flags, please view "Reactions on Warnings" on page 3496 and "Reaction on Errors" on page 3498. If `printf` is used, the C module `ml_mon.c` below the `cmicro/kernel` directory must be compiled also.

Errors are numbered easily by an integer value. If the user's target allows the use of `printf`, then use the C function `xmk_err_text` contained in the `ml_err` module. It is easier to specify error handling in this module rather than directly modifying the `ErrorHandler` function.

Example 611: `Default implementation of ErrorHandler()` **———**

```
void ErrorHandler ( int ErrorNo )
{
  ...
  ...
  ...

 ErrorClass = xmk_GetErrorClass (xmk_TmpErrorNumber);
  #ifdef XMK_WARN_ACTION_HANGUP
    if (ErrorClass==XMK_WARNING_CLASS)
      while (1);
  #endif /* ... XMK_WARN_ACTION_HANGUP */

  #ifdef XMK_ERR_ACTION_HANGUP
    if (ErrorClass==XMK_FATAL_CLASS)
      while (1);
  #endif /* ... XMK_ERR_ACTION_HANGUP */

  #ifdef XMK_WARN_ACTION_PRINTF
    if (ErrorClass==XMK_WARNING_CLASS)
      xmk_MonError (stderr, xmk_TmpErrorNumber);
  #endif /* ... XMK_WARN_ACTION_HANGUP */

  #ifdef XMK_ERR_ACTION_PRINTF
    if (ErrorClass==XMK_FATAL_CLASS)
      xmk_MonError (stderr, xmk_TmpErrorNumber);
  #endif /* ... XMK_ERR_ACTION_HANGUP */

  #ifdef XMK_WARN_ACTION_USER
    if (ErrorClass==XMK_WARNING_CLASS)
      XMK_WARN_USER_FUNCTION( xmk_TmpErrorNumber );
  #endif /* ... XMK_WARN_ACTION_USER */

  #ifdef XMK_ERR_ACTION_USER
    if (ErrorClass==XMK_FATAL_CLASS)
      XMK_ERR_USER_FUNCTION( xmk_TmpErrorNumber );
  #endif /* ... XMK_ERR_ACTION_USER */

  ...
  ...
  ...
```

```
} /* END OF SAMPLE */
```

Example 612: `ErrorHandler()`

```
void ErrorHandler ( int ErrorNo )
{
  /* The user could implement the 3 functions
  ** fatalerror, warning, and information
  */
  switch (ErrNo)
  {
    case  ERR_N_UNKNOWN
          fatalerror(ErrNo);
          break;

    case  .....
          .....
          break;

    default: warning(ERR_N_UNKNOWN); break;
  }
} /* END OF SAMPLE */
```

List of Dynamic Errors and Warnings

Cmicro Kernel Errors

- ERR_N_CREATE_NO_MEM

  This error can happen in the case of using dynamic process creation. When a parent process tries to create another process, the Cmicro Kernel tries to allocate an entry in the queue for the insertion of an internal create signal.

  The error occurs when full queue capacity has been reached.

  Help: Change the size of the signal queue by modifying the macro XMK_MAX_SIGNALS in `ml_mcf.h`.

- ERR_N_CREATE_INSTANCE

  This error happens if there is no dormant instance of a process type which can be (re)used. The Cmicro Package uses fixed upper process instance limits (x, N), where N is to be specified as an integer value. In the C code, all the data of one process instance is represented by one element of an array of the size N.

The error occurs if all instances (i.e. all array elements), of this process type are currently active and the parent tries to create another instance of that type.

- ERR_N_SDL_DISCARD

  A signal was sent to a process not existing (either not yet created or already stopped). The signal is therefore discarded.

- ERR_N_DIVIDE_BY_ZERO

  A division by zero has been detected in the generated SDL code, which is an SDL user error. Possibly, it is helpful to produce an execution trace to localize the error. Normally, it is possible to find such problems within the simulation.

- ERR_N_NO_FREE_SIGNAL

  It is impossible to allocate one more signal instance from the static memory pool. Depending on how the user has defined the signal handling (`XMK_USE_STATIC_QUEUE_ONLY` and `XMK_USE_STATIC_AND_DYNAMIC_QUEUE`), this must be treated either as a warning or as a fatal error.

  It has to be treated as a fatal error, when the macro `XMK_USE_STATIC_QUEUE_ONLY` is set.

  It can be treated as a minor warning, when the macro `XMK_USE_STATIC_AND_DYNAMIC_QUEUE` is set.

  The output operation fails, if it is impossible to allocate one more signal.

  Help: Change the size of the signal queue by modifying the macro XMK_MAX_SIGNALS in `ml_mcf.h` or change the size of predefined dynamic memory pool (the memory pool, that is used when the xAlloc C function is called).

- ERR_N_PARAMETER_MEM_ALLOC

  The output operation fails as a signal with parameters is to be sent but not enough free memory is available for allocation of the signal parameters.

  If the signal parameter's length is greater than the value of XMK_MSG_BORDER_LEN (see "Compilation Flags" on page 3486) the parameters are inserted in a memory area which has to be

allocated. This allocation fails because there is no more memory available.

Help: If using the Cmicro Library's memory functions (flag XMK_USE_SDL_MEM): The size of the memory for allocation can be increased by incrementing the value XMK_MAX_MALLOC_SIZE.

- ERR_N_SYSTEM_SIGNAL

  This error can only happen, if a non specified system signal is sent. Internally, some signals are predefined as system signals. System signals are given priority treatment. For example, for the dynamic process creation, there is a system signal called `XMK_CREATE_SIGNALID` which is automatically defined in `ml_mcf.h` according to the setting of XMK_USE_MORE_THAN_250_SIGNALS.

  To prevent this error situation happening it is strongly recommended not to send an SDL signal with equal or higher priority than that of a system signal (please see the subsection "Scheduling" on page 3469).

- ERR_N_NO_CONT_PRIO

  This error occurs when process priorities are not numbered according to the rules of the preemptive Cmicro Kernel. The numbering must begin from zero incremented consecutively to an upper limit. All numbers between zero and the upper limit-1 are to be used within a #PRIO directive, no number may be omitted. (See MAX_PRIO_LEVELS and `xDefaultPrioProcess` in section "Preemption" on page 3502.

- ERR_N_NO_COR_PRIO

  When the preemptive Cmicro Kernel is selected, the macro MAX_PRIO_LEVELS has to be set correct. As described in MAX_PRIO_LEVELS in section "Preemption" on page 3502, this macro has to contain the amount of process priority levels.

- ERR_N_xRouteSignal

  The function `xRouteSignal`, which is to be filled by the user when signals do not contain a receiver pid, detects an error or was not able

to handle the current signal which is to be routed. See "Bare Integration" on page 3530.

- ERR_N_NO_REC_AVAIL

  If the user is using implicit addressing (output without to) and there are several possible receivers of one type, the Cmicro Kernel tries to assign one of them to this signal. This assignment may fail if there is no possible receiver. If there is more than one, then the first one found will be used in the output of the C function `xmk_Send*`.

- ERR_N_NO_FREE_TIMER

  The error occurs when an SDL process tries to start an instance of a timer and either the timer is unknown or there is no memory available to start a timer instance. To eliminate the first source of error check the SDL compilation. For the second case increase the memory assigned to timers by increasing the value of XMK_TIMERPRIO.

- ERR_N_PID_INDEX

  The Cmicro Kernel uses PIds as index values. Each SDL process in the generated system is numbered by a system-wide unique number. There is an error detected when the index is out of range. This problem most probably lies in the environment which tries to send a signal to a non existing process.

- ERR_N_SEND_TO_NULLPID

  An SDL application tries to send to a NULL - PID. This case normally cannot arise as the Analyzer performs appropriate checks in the dynamic analyses pass. If it occurs, it is most probably that either something is wrong with the environment signalling or possibly with initialization of pid variables.

- ERR_N_TRANS_TIME

  The maximum execution time for one SDL transition was exceeded. This error can only happen if no preemption is used and if the administration of the transition-execution-time is switched on. See "XMK_USE_CHECK_TRANS_TIME" on page 3501.

- ERR_N_xOutEnv

The C function `xOutEnv()` does not handle all necessary signals. It should handle all signals which are sent to the environment. Please view "Return Values of xOutEnv()" on page 3539.

- ERR_N_INIT_SDL_MEM

  If the Cmicro Library's memory functions are used, (XMK_USE_SDL_MEM is defined) the memory for `alloc()` and `malloc()` has to be initialized. This error occurs if a `malloc()` takes place before the memory is initialized. See "ml_mem.c" on page 3566

- ERR_N_SDL_DECISION_ELSE

  An SDL decision without any ELSE branch leads to a fatal error.

- ERR_N_SDL_RANGE

  The index of an SDL array has crossed the range of the SDL array. The definition of the array has to be checked within SDL.

- ERR_N_NO_RCV

  If the flag XMK_USE_RECEIVER_PID_IN_SIGNAL is undefined the function `xRouteSignal()` has to be filled by the user. See "Getting the Receiver of a Signal – Using xRouteSignal" on page 3542. This error code means that there is no receiver defined for the current signal.

- ERR_N_SDL_IMPLICIT_CONSUMPTION

  The receiver process is not expecting the current signal in his current state, so the signal was implicitly consumed.

- ERR_N_PREDEFINED_OPERATOR_CALL

  An error occurred in the call to one of the predefined operators. Unfortunately it is not possible to find out what the reason for this error is without either simulating the system or using the SDL Target Tester or any trace possibility introduced by the user. The error message occurs if a range in the memory is crossed.

- ERR_N_INIT_QUEUE

  The SDL signal queue is not correctly initialized. The error occurs if the user has forgotten to call the C function `xmk_InitQueue()`. This function must be called before `xmk_InitSDL()` may be called.

Please view the section "Implementation of Main Function" on page 3530.

- ERR_N_MEM_PARAM

  This error occurs if the user tries to call to the `xmk_Malloc()` C function with a wrong parameter. It is not allowed to request a memory block of the size 0.

- ERR_N_MEM_NO_FREE

  There is no free block in the memory pool of the Cmicro Memory Management (see "Exported from ml_mem.c" on page 3587). This error can only be detected if the Cmicro Memory Management is used (see the flag "XMK_USE_SDL_MEM" on page 3515).

  The size of the dynamic memory, i.e. the amount of blocks available can be modified by modifying the flag XMK_MAX_MALLOC_SIZE

- ERR_N_MEM_ILLMBLOCK

  A call to the `xmk_Free()` function of the Cmicro Memory Management (see the file "Exported from ml_mem.c" on page 3587) occurred and the given block is an invalid block.

  This error can only be detected if the Cmicro Memory Management is used (see the flag "XMK_USE_SDL_MEM" on page 3515).

- ERR_N_NO_FREE_SIGNAL_DYN

  This error is detected when a new signal is to be allocated dynamically and there is no more free memory available. The Cmicro Kernel starts to create signal instances by using memory allocation in the case that the define XMK_USE_STATIC_AND_DYNAMIC_QUEUE is defined and there is no available signal instance in the predefined pool of static signal instances (see XMK_MAX_SIGNALS).

- ERR_N_NO_FREE_TIMER_DYN

  This error is detected when memory for a new timer instance is to be allocated dynamically and there is no more free memory available. The error only occurs if timers with parameters are used. The Cmicro Kernel starts to create timer instances by using memory allocation in the case that timers with parameters are used in SDL

(XMK_USED_TIMER_WITH_PARAMS is defined in `sdl_cfg.h`) and the amount of statically predefined timer instances (XMK_MAX_TIMER_USER defined with Targeting Expert) is being reached.

- ERR_N_NULL_POINTER_VALUE_USED

  This error occurs within the macros that are generated in order to check null pointer access. These error checks are generated for variables of sort ref, own and oref. If the error occurs, the further behavior of the SDL system is unpredictable.

- ERR_N_UNDEFINED_ERROR

  This error message is implemented to serve the C `switch` statement in use with a default branch. It should not occur during system execution.

SDL Target Tester Errors

- ERR_N_INDEX_SIGNAL

  Under normal circumstances this error should not occur. It is an error detected by the SDL Target Tester when a signal id is out of range. This is only used if signal trace options are modified or asked for.

- ERR_N_ILLEGAL_CMD

  An illegal command was sent to the SDL Target Tester command interface module. This error situation arises due to careless implementation of the function interface and should not occur under normal circumstances.

- ERR_N_TESTER_MESSAGE

  An illegal message was sent to the SDL Target Tester and it is not able to decode the message. A possible error source is an inconsistency in the underlaying protocol on the communications interface.

- ERR_N_LINK_SYNC

  It was not possible to receive the sync byte of the data link frame of the Cmicro Protocol.

- ERR_N_LINK_DEST_BUFFER

A message which is to be decoded is too large for the destination buffer. Possible error sources are an inconsistency in the data definition or the length of information received is wrong.

- ERR_N_LINK_ENC_LENGTH

  An attempt was made to send more than the allowed maximum amount of bytes.

- ERR_N_LINK_ENC_MEMORY

  There is not enough, or no free memory to encode a data link frame of the Cmicro Protocol.

- ERR_N_LINK_NOT_IMPL

  A feature of the data link which is not implemented has been used, for example an attempt to send more than the maximum amount of allowed bytes.

- ERR_N_DATA_LINK

  There is an error on the data link detected.

- ERR_N_DECODE_METHOD

  There is no decoding method defined for a given frame of the Cmicro Protocol.

- ERR_N_RING_WRITE_LENGTH

  There is a ring buffer overflow detected in the data link module.

- ERR_N_TRACE_OUTPUT

  The signal parameter length is greater than the size which can be transmitted.

- ERR_N_RECORD_OUTPUT

  The signal parameter length is greater than the size which can be transmitted.

- ERR_N_RECORD_MAX_EXCEEDED

  The debit counter used for the SDL Target Tester’s record is overflown.

- ERR_N_RECORD_STATE

  The received message cannot be handled in the current recorder state.

- ERR_N_RECORD_CANNOT_CONT

  Because a fatal system error is detected the record session cannot be continued.

Light Integration

A light integration is to be performed if the SDL system including the Cmicro Library should execute within just one operating system task.

The SDL system usually communicates with the environment by using a mailbox or message queue or something similar, which is provided from the operating system. Within the SDL system, the Cmicro scheduler is used. This means that signals that are sent internally in the SDL system are not sent via the mailbox/message queue from the operating system.

Model

The execution model for an SDL system in a light integration can be sketched with the following meta code:

1.

```
/* In the user's initialization task do : */
Initialize memory
Initialize time
Initialize mailbox(es) or message queue(s)
Initialize other resources
Make the SDLTask an operating system task
Wait until the SDLTask has finished initialization
Continue with creating other operating system tasks
```

The model for what the SDL task does in a light integration can be sketched with something like:

2.

```
Initialize SDL Target Tester
Initialize Cmicro SDL queue
Initialize Cmicro Trace options (if wanted)
Initialize SDL and execute start transitions of the
process instances which are to be created statically

Forever do
{
  if there is a signal to be consumed
  {
    Get the next message from the OS queue,
    without suspending the SDL Task
  }
  else, if there are only saved signals
  {
    evaluate the remaining duration for the
```

```
      timer that will expire next

      if there is at least one active timer
      {
        listen on the operating system queue for the
        remaining duration, by using a blocking
        function call
      }
      else
      {
        listen forever to the operating system queue
        by using a blocking function call
      }
      If a message was received, format the message
      and send it as an SDL signal to the SDL system
    }
    If a timeout occured (remaining duration),
    then check all the timers. For each timer that
    has been expired there will be one signal
    put into the SDL queue

    process SDL signal
  }
```

Procedure to Implement the Model

This subsection gives the user the details that he must know for the implementation of the given model. He has to deal with some special macros which must be defined and used. Their names have the prefix `XLI_`.

The user must execute the following steps:

1. Create the module with the C `main()` function.

This module has to contain the C `main()` function, corresponding to the first meta code in the section before. The module should include `ml_typ.h` and the interface file (e.g `component.ifc)`. In this module the definitions of other external tasks can take place. The main function should handle the following steps:

- initialize the memory if required
- initialize the hardware timer
- initialize OS resources (semaphore, message queues,...)
- start the necessary OS tasks, this means also to start the task which represents the SDL state machine.
- wait until all tasks are started

2. Set XLI_LIGHT_INTEGRATION

To set this flag the Targeting Expert can be used. Choose your integration and choose “Support light integration” below Target Library and kernel tab. When the box is chosen three things will happen.

- the compiler flag XLI_LIGHT_INTEGRATION will be set
- the compiler flag XMK_USE_INTERNAL_QUEUE_HANDLING is automatically set
- The macro XLI_INCLUDE is set with the name in the edit line (default is ’li_os_def.h’)

This header file is included in ml_typ.h and must contain a definition of XLI_SDL_TASK_FUNCTION. With the help of this macro, the main function known from the standard Cmicro kernel will be mapped to the OS task function. The body of this function is located in mk_user.c.

3. Define the macros which are needed in the task function.

The task function is corresponding to the second meta code of the model. The following macros can be defined in the header file:

- XLI_TEMP_TASK_VARS

If temporary variables are needed in the task function, they can be declared at this place. It is inserted at the top of the task function.

- XLI_TEMP_QUEUE_VARS

If temporary variables for the communication resources are needed, they can be defined within this macro. It is expanded at the top of the function which queries the queue.

- XLI_OS_TASK_INIT

Any preparations in the task function can be done with this. It follows XLI_TEMP_TASK_VARS at the start of the task function.

- XLI_CREATE_OS_QUEUE

Here the communication resources for the OS task can be created (which ones that will be used depends on the user and the used OS). In the further part of the document the term queue will be used instead of communication resources.

- XLI_START_OS_TASK_SYNC

If a synchronization between the OS tasks is needed, this should be defined with XLI_START_OS_TASK_SYNC.

- XLI_GET_NEXT_MESSAGE_FROM_OS_QUEUE

The function xmk_RunSDL() represents the continuous repeating part of the meta model. Inside of it xmk_QueryOSQueue() is called and it is responsible for the query of the OS queue. If the SDL queue contains signals, XLI_GET_NEXT_MESSAGE_FROM_OS_QUEUE is used to read non-blocking from the OS queue. The task function should not be suspended.

- XLI_LISTEN_ON_OS_QUEUE_WITH_TIMEOUT(TIMEOUT)

If the SDL queue is empty, but at least one timer is active, XLI_LISTEN_ON_OS_QUEUE_WITH_TIMEOUT(TIMEOUT) is used to read from the OS queue with a time-out. The OS queue should be queried as long as the next timer expire. The duration is given with the parameter TIMEOUT.

- XLI_LISTEN_ON_OS_QUEUE_NO_TIMEOUT

If there is no signal in the SDL queue and there is no timer active XLI_LISTEN_ON_OS_QUEUE_NO_TIMEOUT is used to query the OS queue while the next OS message arrives. The SDL task should be suspended.

- XLI_CALL_xInEnv

The content of XLI_CALL_xInEnv is executed after an OS message arrives. It should pass on the content of the received OS message (a signal with parameter) to the SDL system. Normally xInEnv is called to send the signal to the SDL system. The user has to define a data structure which transports the signals from the OS to the SDL system. The name for the data structure can be assigned with XMK_xInEnv_PARTYPE. The name for the parameter of xInEnv can be assigned with XMK_xInEnv_PARNAME.

- XLI_END_OS_TASK_SYNC

This can be defined with something that sign up the other non-SDL tasks about ending of the SDL task.

4. Initialize the environment.

This is usually done in `xInitEnv()`, but probably outside SDL, depending on the needs of the user.

5. Adaptions for sending signals from the SDL task to any other OS task.

Usually, users must fill out the function `xOutEnv()` for sending signals to the environment (another operating system task). It is up to the user what kind of communication is to be used in this direction.

Note:

All necessary resources, e.g. the receivers queue, must have been created before they are used.

6. Adaptions for receiving signals from any other OS task and translation of the signal parameters.

In the SDL task, it is necessary to format the incoming parameters after the message was read from the message queue. Afterwards the signal can be sent to the SDL system with `XMK_SEND_ENV`.

Note:

All necessary resources must have been created before they are used.

7. Implement timers in SDL.

It should be possible to implement hardware timers, like it is described in "Defining the SDL System Time Functions in mk_stim.c" on page 3528. A usual way is to increment a global variable (of type `xmk_T_TIME`) by using an operating system task or similar. A time overrun is automatically detected by the Cmicro Kernel.

8. System shutdown.

There is no default way to shutdown, but usually it is enough to implement the `xCloseEnv` function and call it at an appropriate place.

9. Cmicro Target Tester

The next thing that is remaining is the SDL Target Tester. The instructions which have been given for bare integration, are still valid. The easiest way to use the SDL Target Tester in a light integration is to dedicate a communication interface exclusively for the SDL Target Tester.

Note:

The Target Tester can only used for tracing.

10. Compile and link

The added files must be registered as source files in the Targeting Expert. Please view "Source Files" on page 2935 in chapter 59, *The Targeting Expert* to get information on how to add more files to the list of files to be compiled. The Targeting Expert will automatically add these files to the makefile.

11. Save the settings and press Full Make.

File Structure

Description of Files

The Cmicro Library Functions and Definitions

sdl_cfg.h

This file is automatically generated by the Cmicro SDL to C Compiler into the directory which is currently active. It contains compilation flags used for the automatic scaling of the Cmicro Library and the generated C code. The file must not be edited by the user.

Caution!

The file `sdl_cfg.h` always carries the same name, for each SDL system generated and is stored in the currently active directory (project or working directory). Inconsistencies arise if several systems are to be generated in the same directory. To avoid this situation, it is recommended to use different working directories for each SDL system. Otherwise, unpredictable results at run-time could result, as some required automatic scalable features may/may not have been compiled.

ml_typ.h

This file is the central header file in the Cmicro Package. It contains

- more #includes
- defines, which are of global interest and Cmicro Library internal defines
- typedefs which are of global interest and Cmicro Library internal typedefs
- external declarations which are of global interest and Cmicro Library internal external declarations

sctpredg.h, sctpred.h and sctpred.c

These files contain all definitions necessary to handle SDL data, for example predefined sorts. `sctpred.h` is included in `ml_typ.h`.

ml_err.h

This header file defines all error numbers used by the Cmicro Kernel, the Cmicro Library and the SDL Target Tester.

ml_mem.c

This file contains the dynamic memory management functions from Cmicro. It contains among other functions the C functions `xmk_Malloc()`, `xmk_Calloc()` and `xmk_Free()`. Possible reasons to use this module are:

- Compiler does not support dynamic memory management.
- Dynamic memory management of the compiler does not meet the requirements of the application. This may be the case, if the user wants to use “best fit” instead of “first fit”, the first of which is normally not supported. It is possible to adapt the memory management to ones needs.
- User wants to use other mechanisms of `ml_mem.c`, like profiling. Please view “Dynamic Memory Allocation” on page 3543.

ml_mon.inc

This file is included by the `ml_mon.c` file. It exists only for internal purposes and in order to maintain all the defined errors in the system in a better way.

ml_mon.c

This file contains some help functions which are useful in producing screen outputs. It contains C functions for buffer printouts, SDL PId printouts, displaying error messages, and the display of the current scalings used in the executable. The file should usually be included when compiling the kernel.

ml_*.h

Other header files which contain extern declarations of modules of the Cmicro Library.

The Cmicro Kernel

mk_main.c

This file represents the main-interface to the SDL user. It contains functions which are to be called by the SDL user to integrate the Cmicro Kernel in his application.

- an SDL initialization function `xmk_InitSDL()`
- an SDL system execution function `xmk_RunSDL()`

For SDL Target Tester, there are some more functions to be called during initialization. Please view chapter 67, *The SDL Target Tester*.

> **Note:**
> The SDL queue must in any case be initialized before the SDL system is going to execute. The initialization function is called `xmk_InitQueue()` and is exported from the `mk_queu.c` module.

mk_user.c

This module is not in the Kernel directory, but in the Template directory. It contains function templates or examples which are to be filled out by the user:

`main()`

The C main function contains by default the full access to all Cmicro features.

`ErrorHandler()`

Central error handling routine is called each time an errors occurs.

`WatchdogTrigger()`

Handling of a hardware watchdog.

In earlier versions of the Cmicro Package the functions `xInitEnv()`, `xInEnv()`, `xOutEnv()` and `xCloseEnv()` were include in `mk_user.c`, too. These functions will now be generated by the Targeting Expert and stored in the file `env.c`.

mk_sche.c

This file is the heart of the Cmicro Kernel. It exports those functions which are used in the `mk_main` module and it uses those functions of

other modules which represent the SDL model. The module serves with all the different scheduling policies described later in the subsection "Scheduling" on page 3457.

mk_outp.c

This file contains the SDL operation OUTPUT which is represented by a few functions. There is a C function `xmk_Send()` representing the SDL output operation. In addition, there is a C function `xmk_SendSimple()` which is used when a signal contains no parameters and has no explicitly defined priority. This results in a more compact argument list thus reducing the generated C Code.

mk_queu.c

This module contains the data type "SDL queue" and defines all operations on the SDL queue. The queue handling covers all the aspects of the SDL semantics as far as signals are concerned. There is one function `xmk_InitQueue()` which must be called in the user's `main()` function before the SDL system is going to execute.

mk_tim1.c

This file contains the SDL operations on timers such as set, reset, active, and some help functions used by the Cmicro Kernel. The timer model is described within the subsection "Timers and Operations on Timers" on page 3467. The Cmicro Kernel has to initialize all timers, test for expired timers and reset all timers of a process, when a process instance stops.

mk_stim.c

This file contains some functions which are to be filled up by the user. This is the reason why it is in the Template directory. The functions are used by the Cmicro Kernel to get the system time used in SDL. The contents of this file should be seen as a template. No provision for the connection to hardware timers is provided in the delivered source, as the timer used is application dependent. The user is required to fill out the appropriate functions for the provision of SDL system time. Please view "Defining the SDL System Time Functions in mk_stim.c" on page 3528

mk_cpu.c

This file also contains some hardware specific functions which should be seen as templates. It is also in the Template directory.

mk_*.h

Other header files contain extern declarations of modules of the Cmicro Library. The contents and details of the various header files only need to be known if the user wishes to modify parts of the Cmicro Library.

Functions of the Basic Cmicro Kernel

The list in the following section gives an overview of the functions exported by each module of the Cmicro Library in order to understand the module structure. The functions declared as static are not considered here.

If there is a reference to `xmk_RAM_ptr`, this can be replaced with a * (star) usually. This means that for example the declaration

```
xmk_T_CMD_QUERY_QUEUE_CNF xmk_RAM_ptr qinfo
```

can be replaced with

```
xmk_T_CMD_QUERY_QUEUE_CNF *qinfo
```

which means to refer to a pointer to an object of the type `xmk_T_CMD_QUERY_QUEUE_CNF`.

Exported from env.c

xInitEnv

Parameters:

```
In/Out: -no-
Return: -no-
```

This function is called by the Cmicro Kernel during initialization of the SDL system. The user may include initialization of the environment here.

xInEnv

Parameters:

```
In/Out: -no-
Return: -no-
```

This function is called by the Cmicro Kernel continuously to retrieve signals polled from the environment. Use the Cmicro Kernel function `xmk_Send`* to put signals into the system. The use of this function is not absolutely necessary in the case where the Cmicro Kernel is scaled to preemption and all external Events are put into the SDL system via an Interrupt Service Routines.

xOutEnv

Parameters:

```
In/Out: xmk_T_SIGNAL             sig,
        xmk_T_PRIO               prio,
        unsigned char            data_len,
        void                    *p_data,
        xPID                     Receiver

Return: xmk_OPT_INT
```

This function is called by the Cmicro Kernel if an SDL signal is to be sent to the environment.

> **Note:**
>
> The user has several possibilities to send signals to the environment. Please refer to the subsection about the "Functions of the Expanded Cmicro Kernel" on page 3592.

The function must return with XMK_TRUE, if the Signal was sent to the environment, otherwise it must return with XMK_FALSE.

xCloseEnv

Parameters:

```
In/Out: -no-
Return: -no-
```

This function is called by the Cmicro Kernel during the exit phase of the SDL system. The user may include de-initialization of the environment here.

Exported from mk_user.c

xSDLOpError

Parameters:

```
In/Out: char *xmk_String1 - SDL ADT operators name
        char *xmk_String2 - The reason for failure
Return: -no-
```

This is a function which is to filled up by the user. The function is an central error handling function for ADTs. It is compiled only if XEC-SOP was defined in ml_mcf.h.

ErrorHandler

Parameters:

```
In/Out: int ErrorNo - the given error number
Return: -no-
```

This is a function which is to be filled out by the user. The Cmicro Kernel as well as the SDL application are the main clients of this function.

The user may distinguish between the different errors and define a specific reaction.

The different errors defined in the file `ml_err.h` below the Cmicro Kernel directory.

WatchdogTrigger

Parameters:

```
In/Out: -no-
Return: -no-
```

Description:
If selected this function is called by the Cmicro Kernel each time an SDL transition is executed

> **Caution!**
>
> Be sure, that the time-out used for the Watchdog is longer than the longest SDL Transition (in the case of non preemptive Cmicro Kernel). If the preemptive Cmicro Kernel configuration is used, then the Watchdog Trigger should not be used because the execution time of transitions cannot be calculated.

xRouteSignal

Parameters:

```
In/Out: xmk_T_SIGNAL xmk_TmpSignalID - Signal ID
Return: xPID  Process - PID or xNULLPID
        if no receiver is defined for the given
        signal.
```

Description:
This function is called by the Cmicro Kernel, if SDL signals have no receiver. (undef XMK_USE_RECEIVER_PID_IN_SIGNAL) This might be useful in very small systems in order to spare some RAM memory.The following restrictions apply:

- The SDL System must not contain anything other than the following process declaration. (x,1)
- For each Signal in the SDL System, there is to be only one Receiver process (no signal may be sent to more than one process type).
- no dynamic process creation is used (Create-Symbol is not used)

Note:

Timers are represented as signals, that's because `xRouteSignal` also has to map timers to the receiver process

Hint:

Each signal and timer is represented by a system wide unique integer number.

Exported from mk_main.c

xmk_InitSDL

Parameters:

```
In/Out: -no-
Return: -no-
```

This C function is called by the user before calling the C function `xmk_RunSDL()` and implements the initialization of the whole SDL system, namely Timer, Queue, Processes.

xmk_RunSDL

Parameters:

```
In/Out: -no-
Return: -no-
```

This function operates endlessly unless `XMK_USE_SDL_SYSTEM_STOP` is activated. Then the function is returned if the signal queue is empty or no process is alive. Before processing signals, SDL time-outs are checked and the C function `xInEnv` is called.

xmk_MicroTesterInit

Parameters:

```
In/Out: -no-
Return: -no-
```

This function is used to tell the target configuration to the SDL Target Tester. First the communication interface is initialized, then the startup message is sent to the host. After that the target waits for a "go forever" message from the host.

xmk_MicroTesterDeinit

Parameters:

```
In/Out: -no-
Return: -no-
```

This function is responsible for closing the communication interface to the host after the system has ended.

Exported from mk_sche.c

xmk_StartProcesses

Parameters:

```
In/Out: -no-
Return: -no-
```

This function implements the start-up phase of the SDL system. All static process-instances are created. This means executing the start-transition of all process-instances to be created. For each created process-instance, the first state is set. If configured right, the values SDL_SELF, SDL_PARENT and SDL_OFFSPRING are correctly initialized (only necessary if no semantic check was performed, i.e. if the Analyzer is not used).

xmk_ProcessSignal

Parameters:

```
In/Out: -no-
Return: -no-
```

This function processes an SDL signal and remains in an internal loop, until a signal has been processed or until no signal remains in any input-port in the SDL system.

xmk_CreateProcess

Parameters:

```
In/Out: ProcessID       - ID of the process type
Return: xmk_T_INSTANCE  - created Instance number ID
```

This function tries to create an instance of the given process-type. This can fail, either if the create signal cannot be allocated (no more memory) or if there is no free process instance of that type. E.g. if there is no instance in the `DORMANT` state.

The return value contains the process instance number ID, if one more process instance could be allocated. The return value is set to `xNULLINST` if the creation failed for some reason.

The process instance number is not the same as the process ID. The process ID is calculated from the process ID type plus the process ID instance number.

The generated C code does not use the return value, because the SDL offspring and the parent pid value are stored in the pid tables of the process instance.

xmk_IsAnyProcessAlive

Parameters:

```
In/Out: -no-
Return: xmk_T_BOOL
```

This function checks for any active instance of a process type by searching for instances not in the state XDORMANT.

The function returns with `XMK_TRUE`, if there is an active instance. It returns with `XMK_FALSE` if there is no instance active within the system.

xmk_IfExist

Parameters:

```
In/Out: xPID - Process ID of the process to be
checked
Return: xmk_T_BOOL
```

This function checks if the given PID is valid or not. The return value is `XMK_TRUE` if the given PID is valid. If an instance with this PID does not exist, it returns with `XMK_FALSE`.

xmk_CheckNullPointerValue

Parameters:

```
In/Out: void*
Return: -no-
```

This function checks whether there is a pointer value in the SDL system that is used but has no value. The ErrorHandler is called with the right error message then. The error must be caught in the user's ErrorHandler in order to implement the right reaction on this fatal situation.

xmk_InitPreemptionVars

Parameters:

```
In/Out: -no-
Return: -no-
```

The variables used in preemption are initialized.

xmk_DisablePreemption

Parameters:

```
In/Out: -no-
Return: -no-
```

The variable which stores the preemption status is incremented. A value greater than zero means it is not allowed to perform a context-switch at the moment.

xmk_EnablePreemption

Parameters:

```
In/Out: -no-
Return: -no-
```

The variable which stores the preemption status is decremented if preemption was disabled. If the variable's value is zero after it is decremented, the function `xmk_CheckIfSchedule()` is called.

xmk_FetchHighestPrioLevel

Parameters:

```
In/Out: -no-
Return: xmk_T_PRIOLEVEL
```

This function searches for signals in the priority queue levels. This is done with decreasing priority in order to find the highest priority level at which signals exist. The return value is the highest priority level that contains a signal to process.

xmk_CheckIfSchedule

Parameters:

```
In/Out: -no-
Return: -no-
```

It is checked whether a context switch is admissible. If this is the case the current priority-level is compared with the highest priority-level where a signal exists. Supposing the highest level is higher than the current, a context-switch is performed using `xmk_SwitchPrioLevel()`. This is repeated until the current priority-level is the highest level.

xmk_SwitchPrioLevel

Parameters:

```
In/Out: xmk_T_PRIOLEVEL NewPrioLevel
Return: -no-
```

The global variables for the current priority-level are stored. Afterwards, the function `xmk_ProcessSignal()` is called in order to deal

with the signals on the higher priority level. After returning from this function call the variables for the current priority-level are restored.

With the `NewPrioLevel` the next prio-level to deal with is specified.

xmk_KillProcess

Parameters:

```
In/Out: xPID Process ID
Return: xmk_T_BOOL
```

This function sets a process instance into the state `XDORMANT`, removes all signals directed to it from the queue and resets the local instance data. The return values are `XMK_TRUE` if the call was successfully and `XMK_FALSE` if the process was non-existent, currently running or already in the `XDORMANT` state.

Exported from mk_outp.c

xmk_SendSimple

Parameters:

```
In/Out:
  #ifdef XMK_USE_RECEIVER_PID_IN_SIGNAL
    xmk_T_SIGNAL  sig
    xPID          Receiver
  #else
   xmk_T_SIGNAL   sig
  #endif

Return: -no-
```

This is a simple SDL output function which needs a maximum of only 2 Parameters. Most SDL systems consist of a lot of “normal” Signals without any parameters and no priority. It makes sense to use this simple function whenever possible to spare program code. The signal is put into the linked list of signals by using a default priority.

With the first parameter `sig`, the signal ID of the signal that is to be sent is specified. With the (optional) second parameter, the receiver process ID is specified. If XMK_USE_RECEIVER_PID_IN_SIGNAL is not defined, the user must implement the C function `xRouteSignal()` which is responsible to derive the receiver from the signal ID in that case. Using `xRouteSignal()` is recommended only if the last few bytes must be spared for transferring of signals.

xmk_Send

Parameters:

```
In/Out:
        xmk_T_SIGNAL sig

        #ifdef XMK_USE_SIGNAL_PRIORITIES
          xmk_T_PRIO prio
        #endif

        #ifdef XMK_USED_SIGNAL_WITH_PARAMS
          xmk_T_MESS_LENGTH data_len
          void xmk_RAM_ptr  p_data
        #endif

        #ifdef XMK_USE_RECEIVER_PID_IN_SIGNAL
          xPID Receiver
        #endif

Return: -no-
```

This is, compared to the `xmk_SendSimple` function, a complete SDL output function which needs all possible signal parameters.

With the parameter `sig`, the signal ID is to be specified.

With the `prio` parameter, the signal's priority is to be specified (if conditionally compiled).

With the `data_len`, the number of bytes as the signal's parameters is to be specified. The number of bytes is evaluated by using a `sizeof (C struct)` construct. If the signal carries no parameters, this value must be set to 0 (if conditionally compiled).

With the `p_data` parameter, a pointer to the memory area containing the parameter bytes of the signal is given. The memory area is not treated as dynamically allocated within this function. Because the function copies the parameter bytes, the caller may use any temporary memory (for example memory allocated from the C stack by declaring a C variable). This parameter should be set to `NULL` if no parameter bytes are to be transferred (if conditionally compiled).

With the last parameter `Receiver`, the PID of the receiving process is to be specified (if conditionally compiled).

If the XMK_USE_SIGNAL_PRIORITIES is not defined, the signal priorities which are specified with #PRIO in the diagrams is just ignored. The use of signal priorities is not recommended because the violation of SDL. A few bytes can be spared if signal priority is not used.

The XMK_USED_SIGNAL_WITH_PARAMS is automatically generated into the `sdl_cfg.h` file, from the Cmicro SDL to C Compiler. For tiny systems, if there are no SDL signals with parameters specified, this is undefined. It will reduce the amount of information which is to be transferred for each signal with a few bytes.

If XMK_USE_RECEIVER_PID_IN_SIGNAL is not defined, the user must implement the C function `xRouteSignal()` which is responsible to derive the receiver from the signal ID in that case. Using `xRouteSignal()` is recommended only if the last few bytes must be spared for transferring of signals.

XMK_SEND_ENV

Parameters:

```
In/Out: xPID Env_ID

        xmk_T_SIGNAL sig

        #ifdef XMK_USE_SIGNAL_PRIORITIES
          xmk_T_PRIO prio
        #endif

        #ifdef XMK_USED_SIGNAL_WITH_PARAMS
          xmk_T_MESS_LENGTH data_len,
          void xmk_RAM_ptr  p_data
        #endif

        #ifdef XMK_USE_RECEIVER_PID_IN_SIGNAL
          xPID Receiver
        #endif

Return: -no-
```

This function is to be called when a signal is to be sent into the SDL system, e.g. within the users `xOutEnv()` function. It must be called with one more parameter than the `xmk_Send()` function, which is the first parameter `Env_ID`.

The macro internally uses some variables which are to be declared before the macro can be used. For example in the xOutEnv function the `XMK_SEND_TMP_VARS` macro must be introduced for declaring these variables.

The function is implemented as a macro in C.

xmk_Determine_Receiver

Parameters:

```
In/Out: xmk_T_PROCESS proc_type - process type ID
```

```
Return: xPID                        - process PID
```

This function is used in the context of SDL output in generated C code. The function determines if any instance with the given process type-ID is available to receive the signal. If no instance can be found, `xNULLPID` is returned.

Exported from mk_queu.c

xmk_InitQueue

Parameters:

```
In/Out: -no-
Return: -no-
```

This function initializes the signal queue. It must be called before any other Cmicro Kernel function, e.g. before `xmk_InitSDL()`. All relevant pointers are initialized. All signal-elements are put into the free-list. The SAVE-state of all signals is set to false.

xmk_FirstSignal

Parameters:

```
In/Out: -no-
Return: xmk_T_MESSAGE*
```

The first signal in the current queue which is the one with the highest priority, is copied to the pointer of the currently treated signal and returned to the caller.

The function returns a pointer to the first signal in the queue or `NULL`, if there are no signals in the queue.

xmk_NextSignal

Parameters:

```
In/Out: -no-
Return: xmk_T_MESSAGE*
```

The signal following the current signal is copied to the current signal. If there are no more signals, NULL is returned.

xmk_InsertSignal

Parameters:

```
In/Out: xmk_T_MESSAGE xmk_RAM_ptr p_Message
Return: -no-
```

This function puts a signal into the queue. The position depends on only the signal priority, if specified.

With the parameter `p_Message` a pointer to the signal which is to be inserted is given.

xmk_RemoveCurrentSignal

Parameters:

```
In/Out: -no-
Return: -no-
```

The signal which was currently processed is removed from the queue and inserted into the list of free signals.

xmk_RemoveSignalBySignalID

Parameters:

```
In/Out: xmk_T_SIGNAL SignalId
Return: -no-
```

This function is not used if timers with parameters are in the system (`XMK_USED_TIMER_WITH_PARAMS` macro is defined).

Signals of a given signal-code sent to the current process are removed from the queue and inserted in the list of free signals. The signal currently being processed must not be removed, as it is necessary for the current actions. It is only removed after processing is complete.

With the parameter `SignalID`, the signal ID of the signals which are to be removed for the currently active process instance is specified.

xmk_RemoveTimerWithParameter

Parameters:

```
In/Out: xmk_T_SIGNAL SignalId
In    : SDL_Integer  TimerParValue
Return: -no-
```

This function is only compiled if timers with parameters are in the system (`XMK_USED_TIMER_WITH_PARAMS` macro is defined).

This function has the same purpose as the above `xmk_RemoveSignalBySignalID` but, in addition, must look to the timer's parameter.

xmk_IsTimerInQueue

Parameters:

```
In/Out: xmk_T_SIGNAL TimerID
#ifdef XMK_USED_TIMER_WITH_PARAMS
In    : SDL_Integer  TimerParValue
#endif
Return: -no-
```

This function checks if the given timer is in the signal queue. If timers with parameters are used, the `TimerParValue`, which is conditionally compiled, is checked also.

xmk_RemoveSignalsByProcessID

Parameters:

```
In/Out: xPID ProcessID - PID of Process
Return: XMK_TRUE   - Signal removed
        XMK_FALSE  - no Signal removed
```

All signals addressed to a specific process are removed by calling this Function.

With the parameter `ProcessId`, the PID of the process for which all the signals are to be removed from the queue is specified.

xmk_AllocSignal

Parameters:

```
In/Out: -no-
Return: xmk_T_MESSAGE xmk_RAM_ptr
```

According to the principle chosen by the user (the user has to choose between `XMK_USE_STATIC_QUEUE_ONLY` and `XMK_USE_STATIC_AND_DYNAMIC_QUEUE`), and if possible, an initialized signal instance is returned. The signal instance is then either taken from the static memory pool or from the dynamic memory pool.

Allocation from the dynamic memory pool takes place by calling the `xAlloc` C function.

If `XMK_USE_STATIC_QUEUE_ONLY` is set, the `ErrorHandler` is called with the error "`ERR_N_NO_FREE_SIGNAL`". This indicates that no more memory is available to create one more signal instance, and the user may react appropriately in the `ErrorHandler` C function.

Otherwise, if `XMK_USE_STATIC_AND_DYNAMIC_QUEUE` is set, and if it is impossible to allocate one more instance from the predefined static memory pool, the `ErrorHandler` is called with the error "`ERR_N_NO_FREE_SIGNAL`". This indicates that dynamic memory allocation is started now, and the user may react appropriately in the `ErrorHandler` C function (for example, he might want to print out a warning message). If it is impossible to allocate one more signal from the dynamic memory pool, the `ErrorHandler` is called with the error "`ERR_N_NO_FREE_SIGNAL_DYN`". The user may then also decide what to do in the `ErrorHandler` C function.

A pointer to the signal that was allocated is returned usually, or `NULL`, if there is no space left to allocate one more signal.

xmk_FreeSignal

Parameters:

```
In/Out: xmk_T_MESSAGE xmk_RAM_ptr p_Message
Return: -no-
```

The given signal is de-allocated again.

If the signal was allocated from the static memory pool, it is returned to that pool by initializing it and inserting it into the free list at the first position.

Otherwise, if the signal was allocated from the dynamic memory pool, it will be returned to that pool (by calling the `xFree` C function).

The parameter `p_Message` must point to the signal that is to be initialized.

xmk_TestAndSetSaveState

Parameters:

```
In/Out: xmk_T_STATE State
Return: xmk_T_BOOL
```

This function checks whether a signal's SAVE-state is set or not. In testing, the SAVE-state is set to `TRUE` or `FALSE`.

With the parameter `State`, the value of the process' current state is to be specified.

The function returns with `XMK_TRUE`, if the given state equals the signal's SAVE-state, but returns with `XMK_FALSE` if the given state differs from the signal's save state.

xmk_QueueEmpty

Parameters:

```
In/Out: -no-
Return: -no-
```

This function tests whether there is at least one signal remaining in the queue(s) or not.

The function returns `XMK_TRUE`, if there are no signals in the queue, but returns `XMK_FALSE` if there is at least one signal in the queue. It does not matter if the signal is a saved signal or not.

When the Cmicro Kernel is configured for preemption, all the queues of the different priority levels are checked.

Exported from mk_tim1.c

xmk_InitTimer

Parameters:

```
In/Out: -no-
Return: -no-
```

All initialization of timers is performed within this function, which is called during SDL system start. It initializes some pointers and the free list of timers.

xmk_TimerSet

Parameters:

```
In/Out: xmk_T_TIME   time
        xmk_T_SIGNAL sid
#ifdef XMK_USED_TIMER_WITH_PARAMS
In    : SDL_Integer  TimerParValue
#endif
Return: -no-
```

This function activates an instance of a timer with the given “signal ID” value and the given time. If timers with parameters are used, the `TimerParValue`, which is conditionally compiled, is used also (the value is set to 0 outside this function call if it is not a timer with parameter).

Working principles:

- If a timer instance of this type is already running, this will be deactivated i.e. reset and then set.
- If no free timer is available, the `ErrorHandler()` is called.
- If all is satisfactory, a timer instance is created for the currently running process.

The first parameter `time` specifies the time at which the timer should expire. The call to SDL now is performed in generated C code.

The second parameter `sid` specifies the ID of the timer that is to be started.

If a timer instance of this type is already running, this will be deactivated. If no free timer is available, the ErrorHandler is called. After all these checks a timer instance is created for the currently running process.

xmk_TimerReset

Parameters:

```
In/Out: xmk_T_SIGNAL sid
#ifdef XMK_USED_TIMER_WITH_PARAMS
In    : SDL_Integer  TimerParValue
#endif
Return: -no-
```

This function resets the timer with the given “signal ID” value, if it is active and set by the currently running process. If an active timer instance is found, then the timer is inserted into the free-list. If timers with parameters are used, the `TimerParValue`, which is conditionally compiled, is used also (the value is set to 0 outside this function call if it is not a timer with parameter).

The parameter `sid` specifies the ID of the timer which is to be reset.

xmk_TimerActive

Parameters:

```
In/Out: xmk_T_SIGNAL sid
#ifdef XMK_USED_TIMER_WITH_PARAMS
In    : SDL_Integer  TimerParValue
#endif
Return: xmk_T_BOOL
```

This function checks if a timer with the given ’Signal-ID’ value is active in the current running process. If timers with parameters are used, the TimerParValue, which is conditionally compiled, is used also (the value is set to 0 outside this function call if it is not a timer with parameter).

The parameter `sid` specifies the ID of the timer which is to be checked if it is active.

The function returns with `XMK_TRUE` if the timer is active in the process that is currently running, it returns `XMK_FALSE` otherwise.

xmk_ChkTimer

Parameters:

```
In/Out: -no-
Return: -no-
```

The main intention of this function is to check, if any of the timers which are in the list of active timers is expired. There is a list of active timers in which the active timers are stored in the order of their expiry.

This means that the timer that expires first stands in front of the list.

If a timer was recognized as expired, a timer signal is sent to the owning process instance, the timer is removed from the list of active timers and restored into the list of free timers.

If there is no timer that can expire, the function returns immediately.

Another task for this function is to check if the global system time is overrun. If this case occurs, the `xmk_ChkTimer()` function will readjust the global system time (by calling `xmk_SetTime()` from `mk_cpu.c`) and will readjust all the timers in the active list.

xmk_ResetAllTimer

Parameters:

```
In/Out: xPID pid
Return: -no-
```

All active timers of the given process instance are reset. This will occur if a process instance stops.

With the `pid` parameter the process instance is addressed.

Exported from mk_stim.c

xmk_InitSystime

Parameters:

```
In/Out: -no-
Return: -no-
```

This function initializes the hardware timer and is to be filled out by the user.

xmk_DeinitSystime

Parameters:

```
In/Out: -no-
Return: -no-
```

This function de-initializes the hardware Timer and is to be filled out by the user.

xmk_SetTime

Parameters:

```
In/Out: xmk_T_TIME time
Return: -no-
```

This function sets the system time to the given value `time` and is to be filled out by the user.

xmk_NOW

Parameters:

```
In/Out: -no-
Return: xmk_T_TIME
```

This function is used by

- the Cmicro Kernel to calculate if a timer has expired and this is only the case if absolute time is used by the selected timer model.
- SDL applications to retrieve the current time

The current SDL system time is returned.

The function is to be filled out by the user.

Exported from ml_mem.c

xmk_MemInit

Parameters:

```
In/Out: char* _mem_begin
        char* _mem_end
Return: -no-
```

This function is to be called by the user, before dynamic memory management can be used. The user has to specify the beginning and the end of the area to be used for dynamic memory management. The function should be called before the first call to `xAlloc()`, e.g. as the first statement in the user's `main()` function.

xmk_Malloc

Parameters:

```
In/Out: unsigned long rsize
Return: void *
```

This function allocates one block of memory from the dynamic memory pool. It uses a first fit policy.

xmk_Calloc

Parameters:

```
In/Out: unsigned long RequestedSize
Return: void*
```

This function allocates one block of memory from the dynamic memory pool. It is the same as `xmk_Malloc()` but sets the allocated block to zero.

xmk_Free

Parameters:

```
In/Out: void *mem
Return: void
```

This function is the counterpart of `xmk_Malloc()`. A memory block is de-allocated again.

xmk_Memshrink

Parameters:

```
In/Out: void  *pMemBlock
        unsigned long NewSize
Return: -no-
```

This function is an extension of the compared with the standard of dynamic memory management supported by usual C compilers. It allows the user to shrink down a memory area which was previously requested with `xmk_Malloc()`.

memset

Parameters:

```
In/Out: char *p
        char val
        int length-
Return: -no-
```

This function is a template for the `memset()` implementation.

Caution!

Take care when the preemption policy is utilized.

memcpy

Parameters:

```
In/Out: char *dest
        char *source
        int length
Return: -no-
```

This function is a template for the `memcpy()` implementation.

Caution!

Take care when the preemption policy is utilized.

Exported from ml_mon.c

xmk_GetErrorClass

Parameters:

```
In/Out: xmk_T_ERR_NUM EightBitNumber
Return: int
```

This function returns the error class assigned to the given error number given by the caller. The error class can be one of `XMK_FATAL_CLASS` or `XMK_WARNING_CLASS`. This can be used to classify the predefined error and warning messages and to react properly if such a message occurs.

xmk_MonError

Parameters:

```
In/Out: FILE *fp
        int nr
Return: -no-
```

This function is used to evaluate an ASCII error text from an errornumber given by the caller. The first parameter `fp` must contain a pointer to a valid (i.e. file is opened) file descriptor, for example it could be `stdin`, `stdout`, or a file which was opened with `fopen`. The second parameter `nr` contains one of the possible error numbers defined in `ml_err.h`.

xmk_MonPID

Parameters:

```
In/Out: char *ostring
        xPID pid
Return: -no-
```

Used for test purposes, free use.

xmk_MonHexSingle

Parameters:

```
In/Out: char *p_text
        unsigned char *p_adress
        int length
Return: -no-
```

Used for test purposes, free use.

xmk_MonHex

Parameters:

```
In/Out: char *p_text
        unsigned char *p_adress
        int length
Return: -no-
```

Used for test purposes, free use.

xmk_MonHexAsc

Parameters:

```
In/Out: -no-
Return: -no-
```

Function for test purposes, free use.

xmk_MonConfig

Parameters:

```
In/Out: -no-
Return: -no-
```

Function for test purposes, free use.

Exported from mk_cpu.c

xmk_PutString

Parameters:

```
In/Out: char * Param1
Return: -no-
```

A template for how to print out a character string. The character string must be terminated with "`\0`".

The function must return immediately, i.e. may not be blocking on the output device. If the user does not care about this restriction, the correct function of other program parts cannot be guaranteed.

xmk_GetChar

Parameters:

```
In/Out: -no-
Return: int
```

This function checks if the user has pressed a key on the keyboard (unblocked). The function must return immediately, i.e. may not be blocking on the input device. If the user does not care about this restriction, the correct function of other program parts cannot be guaranteed.

xmk_printf

Parameters:

```
In/Out: char* format
        format string
        other parameters
Return: -no-
```

This function is called from any place in the Cmicro Library, Cmicro Kernel or generated C code, if the `printf` functionality is compiled with at least one of the `XMK_ADD_PRINTF*` defines.

The function must return immediately, i.e. may not be blocking on the output device. If the user does not care about this restriction, the correct function of other program parts cannot be guaranteed.

The function can be used for ANSI C compilers only, because optional argument lists are used like in the `printf` function of the standard C library.

The return value has no meaning and is introduced just for compatibility with `printf`.

xAlloc

Parameters:

```
In/Out: xptrint  Size
Return: void *
```

This function is called from any place in the Cmicro Library, Cmicro Kernel, SDL Target Tester or generated C code, if memory is to be allocated dynamically. The user may choose between the dynamic memory allocation functions from the C compiler or operating system or the dynamic memory allocation functions from Cmicro.

The return value points to the allocated buffer or is `NULL` if the operation was unsuccessful.

xFree

Parameters:

```
In/Out: void **
Return: -no-
```

This is the counterpart of `xAlloc()`. The function is called when a memory block that has been allocated with `xAlloc()` can be de-allocated again. The parameter is the address of the pointer to the allocated buffer.

Example 613 Using the `xFree` function ––––––––––––––––––––––

```
unsigned char *ptr;
ptr = xAlloc(100);
xFree (&ptr);      /* NOTE: Not xFree(ptr); */
```

––

Functions of the Expanded Cmicro Kernel

The expanded Cmicro Kernel offers additional functionality that is usually not necessary for target applications. Extended functionality is for example required by the SDL Target Tester, but may also be required from the user if it comes to target integration.

It is not absolutely necessary to have knowledge of these functions but knowledge of the functions proves useful when it comes to debugging and testing an application.

Functions for Internal Queue Handling

Exported from mk_queu.c

xmk_SaveSignalsOnly

Parameters:

```
In/Out: -no-
Return: xmk_T_BOOL
```

This function tests whether there only are SAVE signals contained in the queue. This can be used in integrations with operating systems.

The function returns `XMK_TRUE`, if there are SAVE signals only in the queue, otherwise it returns `XMK_FALSE`.

Exported from mk_tim1.c

xmk_NextTimerExpiry

Parameters:

```
In/Out: xmk_T_TIME * RemainingTime
Return: xmk_T_BOOL
```

This function looks for the remaining time of the timer that expires next. This is useful for operating system integration. If there is no active timer, the function returns `XMK_FALSE` and the returned parameter `RemainingTime` is set to 0.

If there is an active timer, the function returns with `XMK_TRUE` and the returned parameter `RemainingTime` contains the time at which the timer expires next. `XMK_TRUE` means there is a timer active and the remaining time is returned in the parameter. `XMK_FALSE` means that there is no timer active and returned parameter is set to 0.

Functions to get System Information

Exported from mk_sche.c

xmk_GetProcessState

Parameters:

```
In/Out  xPID pid
Return: xmk_T_STATE
```

The function evaluates the current SDL state of the SDL process `pid` with the given process PID and returns it to the caller. During the SDL system start-up, one possible return value might be `XSTARTUP`. This means that a process will be started during system start-up. During normal execution and system start-up, the return value `XDORMANT` may be returned. This means that an SDL process instance is either stopped or has never been created dynamically (created).

xmk_SetProcessState

Parameters:

```
In/Out: xPID pid,
        xmk_T_STATE state
Return: xmk_OPT_INT
```

The function sets the SDL-state of the process `pid` that is addressed with the first parameter to the given value `state` within the second parameter. Note, that the value is not checked, because Cmicro does not store any information about this in the generated transition tables.

xmk_GetProcessInstanceData

Parameters:

```
In/Out: xPID pid
Return: void *
```

The function returns the address of the process instance data of the given process-PID, or XNOTEXISTENT, if the pid is not existent.

xmk_QueryQueue

Parameters:

```
In/Out: xmk_T_CMD_QUERY_QUEUE_CNF * qinfo
Return: -no-
```

This function evaluates the current SDL-Queue-State and returns some information to the caller:

- information about the size of the Q (maximum amount of entries)

- information about traffic load, measured until now. (Users can directly use this to scale the queue. This information is valuable in the case, where the maximum traffic load is reached during execution.)
- Information about the number of entries that are currently in the queue.
- A pointer to the physical address of the queue.

Exported from mk_tim1.c

xmk_QueryTimer

Parameters:

```
In/Out: xmk_T_CMD_QUERY_TIMER_CNF *tinfo
Return: -no-
```

This function evaluates the current state of the SDL-Timer handling and returns it to the caller. The following information is contained in the C structure that is returned:

- Information about the size of the timer list entries.
- Information about traffic load, measured until now. (Users can directly use this to scale the timer list. This is valuable in the case, when the maximum traffic load is reached during execution)
- The current number of entries in the timer list.
- A pointer to the physical address of the timer list.

xmk_FirstTimer

Parameters:

```
In/Out: -no-
Return: xmk_T_TIMER *
```

The function returns a pointer to the first active timer. If there is no active timer in the queue, `NULL` will be returned.

xmk_NextTimer

Parameters:

```
In/Out: -no-
Return: xmk_T_TIMER *
```

The function returns a pointer to the next active timer in the list of active timers. If there is no or one active timer in the queue, `NULL` will be returned.

Exported from ml_mem.c

xmk_GetOccupiedMem

Parameters:

```
In/Out: -no-
Return: size_t
```

This function returns the currently occupied amount of memory from the pool. The pool size is defined with `XMK_MAX_MALLOC_SIZE`.

xmk_GetFreeMem

Parameters:

```
In/Out: -no-
Return: size_t
```

This function returns the amount of available memory from the pool. This means that it returns the difference between the size of the pool (`XMK_MAX_MALLOC_SIZE`) and the currently occupied memory.

xmk_CleanPool

Parameters:

```
In/Out: -no-
Return: int
```

This function reinitialize the memory pool to free memory leaks. The memory pool can only be cleaned if there are no allocated blocks left.

Alternative Function for sending to the Environment

Exported from mk_outp.c

xmk_EnvSend

Parameters:

```
In/Out:
        xmk_T_SIGNAL sig

        #ifdef XMK_USE_SIGNAL_PRIORITIES
          xmk_T_PRIO prio
        #endif

        #ifdef XMK_USED_SIGNAL_WITH_PARAMS
          xmk_T_MESS_LENGTH data_len
          void xmk_RAM_ptr  p_data
        #endif

        #ifdef XMK_USE_RECEIVER_PID_IN_SIGNAL
```

```
              xPID Receiver
            #endif

   Return: -no-
```

This function is an alternative to the standard function XMK_SEND_ENV for putting signals into the SDL system in external C code. The function is useful when signals are to be sent in interrupt service routines directly. If interrupt service routine is used and can result in signal sending the user need to manually define flag XMK_USE_INTERRUPT_SERVICE_ROUTINE to enable more critical paths in kernel. The function is shorter than XMK_SEND_ENV, but performs not so many error checks. It is especially not checked, if the receiver process ID exist. Usually there is also no call to functions which produce trace output because this would lead to problems in interrupt service routines. But for some error situations like either

- no free signal available
- no more memory available

the ErrorHandler() C function, which is to be implemented by the user, is called.

With the parameter sig, the signal ID is to be specified.

With the prio parameter, the signal's priority is to be specified (if conditionally compiled).

With the data_len, the number of bytes as the signal's parameters is to be specified. The number of bytes is evaluated by using a sizeof (C struct) construct. If the signal carries no parameters, this value must be set to 0 (if conditionally compiled).

With the p_data parameter, a pointer to the memory area containing the parameter bytes of the signal is given. The memory area is not treated as dynamically allocated within this function. Because the function copies the parameter bytes, the caller may use any temporary memory (for example memory allocated from the C stack by declaring a C variable). This parameter should be set to NULL if no parameter bytes are to be transferred (if conditionally compiled).

With the last parameter Receiver, the PID of the receiving process is to be specified (if conditionally compiled).

If XMK_USE_SIGNAL_PRIORITIES is not defined, the signal priorities which are specified with #PRIO in the diagrams is just ignored. The use of signal priorities is not recommended because the violation of SDL. A few bytes can be spared if signal priority is not used.

The XMK_USED_SIGNAL_WITH_PARAMS is automatically generated into the `sdl_cfg.h` file, from the Cmicro SDL to C Compiler. For tiny systems, if there are no SDL signals with parameters specified, this is undefined. It will reduce the amount of information which is to be transferred for each signal with a few bytes.

If XMK_USE_RECEIVER_PID_IN_SIGNAL is not defined, the user must implement the C function `xRouteSignal()` which is responsible to derive the receiver from the signal ID in that case. Using `xRouteSignal()` is recommended only if the last few bytes must be spared for transferring of signals.

Technical Details for Memory Estimations

Allocating Dynamic Memory

Introduction

This section shows when and how dynamic memory allocation is used in the Cmicro Package. It shows,

- how dynamic memory is allocated
- how to estimate the memory requirements for an application

The Cmicro Package uses a form of dynamic memory management for the following objects:

- processes
- signals
- timer
- some of the predefined sorts like charstring, ASN.1 sorts and generators

However, real dynamic memory management is used only in one case, namely for SDL signals, if a signal carries parameters with more than a few bytes.

This means that the Cmicro Kernel has its own memory management to handle processes, signals, and timers. This is done in such a way that each of these 3 objects are managed separately. For each of these 3 objects, a separate fixed memory area is reserved during compilation time, i.e. the area that handles processes cannot be reused to handle timers. This seems to be a restriction but in many micro controller applications users have to fix an upper limit of processes, signals and timers which can be handled in parallel during run time.

Processes

Processes are handled without dynamic memory allocation functions. The user has to specify an upper limit of process instances in the SDL diagram separately for each SDL process. For each process instance, there is a variable which is statically allocated.

Signals with and without Parameters

Cmicro signals are both ordinary SDL signals as well as timer signals. Where timers are implemented without parameters and can therefore be regarded as signals without parameters.

- No `malloc`/`free` for timers
- No `malloc`/`free` for signals without parameters
- No `malloc`/`free` for signals with parameters if less than or equal to `XMK_MSG_BORDER_LEN` bytes parameters are to be transferred.
- malloc and free are used for signals with parameters if more than `XMK_MSG_BORDER_LEN` bytes parameters are to be transferred.
- `XMK_MSG_BORDER_LEN` bytes is a macro defined in the manual configuration file `ml_mcf.h`. It can be set to any value, i.e. zero or the maximum number of signal parameters to be handled in the system.

Timers

For timers, no malloc and free functions are used. The Cmicro SDL to C Compiler evaluates the amount of timers in the system and generates a C constant, which is then used in the Cmicro Kernel to define an array for timers.

Chapter 66 The Cmicro Library

Chapter

67 *The SDL Target Tester*

The SDL Target Tester is a tool which can be used to find errors and illegal or unwanted behavior within a target system. It allows the programmer to follow and to reproduce the execution flow of the SDL system (running on the target) on the host.

The strength of the SDL Target Tester is the minimum amount of storage it occupies on the target. With the exception of certain traces, the SDL Target Tester's memory requirements do not increase proportionally with system size. That means, that the memory amount is a compiler specific constant. If the memory is scarce it is also possible to reduce the memory by reducing the functionality.

In order to use the SDL Target Tester, the system must be developed with the Cmicro SDL to C Compiler (see chapter 65, *The Cmicro SDL to C Compiler*) and the Cmicro Library (see chapter 66, *The Cmicro Library*).

The host and target systems can be connected by any type of communications link, as any kind of communications drivers can easily be connected to the SDL Target Tester system.

Introduction

The SDL Target Tester offers the following features:

- Tracing of internal and external SDL events with the Tracer
- Generating MSC Traces with the Tracer
- Trace of system errors
- Debugging facilities
- Profiling
- Open interface to connect other tools
- Scalable functionality
- Error reproduction using the Recorder

Note:

The SDL Target Tester's Record and Play functions are only available if a Cmicro Recorder license is available.

The SDL Target Tester will be executed in a distributed fashion, i.e. some parts of the SDL Target Tester run on a host machine, while other parts run together with the Cmicro Kernel and the SDL application on the target machine, for example a micro controller. A prerequisite is a communications link between the host and target systems.

The SDL Target Tester – An Overview

Prerequisites

The SDL Target Tester is an optional part of the Cmicro Package and is delivered when specifically ordered. It is an addition to the Cmicro Library and contains the following parts:

- the host tool chain (see "Using the SDL Target Tester's Host" on page 3610)
- the target library (see "The Target Library" on page 3676)

The SDL Target Tester's host executable is a tool chain which consists of the executables sdtmt, sdtmtui, sdtmpm and sdtgate and sockgate. The sdtgate module contains the V.24 and sockgate the TCP/IP implementation for connecting host and target. All open interface functionality is utilized here.

The target library is mandatory when the SDL Target Tester is ordered. It must be used together with the Cmicro Library on the target. By using the open interface, it is possible to connect user specific tools to the SDL Target Tester's target library.

This diagram gives an overview of the SDL Target Tester functionality:

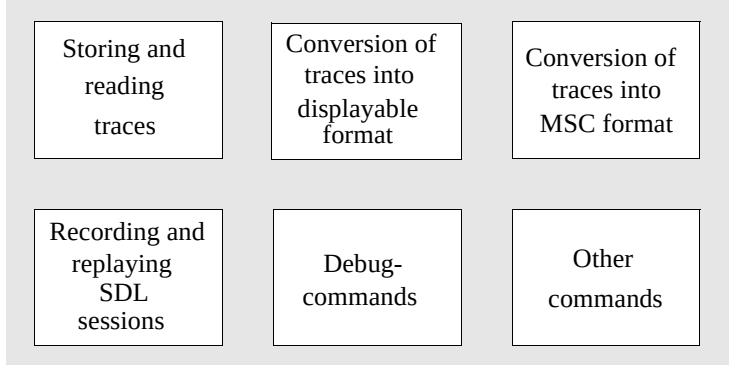


Figure 599: Functionality of the SDL Target Tester

The following components, as seen from the user's point of view, represent the SDL Target Tester:

- The Cmicro Tracer - trace of SDL and system events.
- The Cmicro Recorder - recording and replaying SDL sessions.
- The command and debug interface.

The Cmicro Tracer is introduced in the subsection "Following the Execution Flow – The Cmicro Tracer" on page 3604.

The Cmicro Recorder is introduced following the subsection "Reproduction of Errors – The Cmicro Recorder" on page 3606.

The command and debug interface is introduced in the subsection "Commands for the Host and Debugging Facilities" on page 3609.

The following subsections give an overview of how to use the SDL Target Tester. A detailed description of the necessary preparations and adaptations will be given in the sections itself.

- "Using the SDL Target Tester's Host" on page 3610"
- "Graphical User Interface" on page 3656
- "The Target Library" on page 3676
- "Connection of Host and Target" on page 3688
- "More Technical Descriptions" on page 3730.

These sections follow after the introduction subsections.

Following the Execution Flow – The Cmicro Tracer

General

The Cmicro Tracer transmits information about the execution flow of the SDL system on the target. Information traced includes the signals sent and received by processes and the SDL states and symbols entered in execution flow. A filter can be defined so that only partial trace information is created for particular process instances or for particular SDL constructs. The tracer can be connected with the MSC Editor on the host, thus allowing observation of dynamic signal trace between processes in the system or from the environment to the system on the target.

Tracing SDL Events

The following SDL events can be traced.

State	The SDL nextstate operation
Input	The SDL input operation
Save	The SDL save operation
Output	The SDL output operation
Create	The SDL create operation
Stop	The SDL stop operation
Static Create	Used at system start, appears for each statically created process
Dynamic Create	Used at start of a dynamically created process
Decision	Trace of SDL action decision
Task	Trace of SDL action task
Procedure	Trace of SDL procedure call
Set	Trace of SDL action: Set timer
Reset	Trace of SDL action: Reset timer
Signalparams	Used to switch on/off trace of signal parameters
Implicit consumption	Trace of SDL implicit transition
Discard	Trace of SDL discard

For each symbol in each process in the system the trace can be separately switched on or off.

Please consult the subsection "The API on Target" on page 3679 for information about how to set the different options.

Tracing Other Events

The following system events can be traced.

Schedule	Used to trace scheduling events.
Error	Used to trace System errors, which are detected either within the generated code, within the Cmicro Kernel or within the SDL Target Tester.
Showprio	Used to trace the event, when the scheduling changes to a different priority level.

Each system event can be separately switched on or off.

Please consult the subsection "The API on Target" on page 3679 for information about how to set the different options.

Display of Traces in the MSC Editor

By using the SDL Target Tester, it is possible to create Message Sequence Charts with the MSC Editor. Two modes are of special interest:

- Generating an MSC trace during an SDL session.
- Conversion of a binary file into the MSC format.

The first case, however, only makes sense if the host has enough time to handle and display all the traces created during the run-time of the SDL system. Alternatively there might be SDL systems which do not have these timing constraints. For example, the SDL system may wait until the MSC trace is completed.

The second possibility does not need that much processor power on the host site. In practice, this method is more useful.

The method with using the Cmicro Recorder is the one with the best real-time properties. In comparison to the trace, the Cmicro Recorder stores only a reduced subset of information in a compact format.

Reproduction of Errors – The Cmicro Recorder

Note:

The SDL Target Tester's Record and Play functions are only available if a Cmicro Recorder license is available.

General

Using the Cmicro Recorder, the user can “record” the actions taken within an SDL system running on the target and save these to a file on the host machine. This file can later be “played” in order to run the system through the same state changes, so that the same actions are performed. In this way error situations, perhaps recorded by a client or a developer at another location, can be reproduced. In the following, a general description is given. A more detailed description of the internal procedures can be found in the subsection ““Type and Amount of Stored Information” on page 3733”. The subsection ““General Restrictions on Record and Play” on page 3737”may also provide more information about how it works.

Recording an SDL Session

Use of the Cmicro Recorder makes sense whenever an SDL system is executed in real-time on the target in combination with a host.

The amount of information transferred via the communications link and stored into a file on the hard disk is kept small and compact, so that the real-time properties are not influenced. Only information is stored, which is necessary in order to replay an SDL session later. It is the communication of the environment to the SDL system, and the expiration of timers, which is recorded.

Caution!

The real-time properties, and correspondingly the behavior of the SDL system may be affected when recording. The user has to ensure that there is enough time to process the functions of the Cmicro Recorder, and to transfer all the information via the communications link. In general, it is a good idea to use a high performance communications link with a large transmit buffer and a host machine, which is able to handle all recorded events at any time during the SDL execution.

It is not possible to receive information about the inner events of the recorded SDL system, by using the recorder only. The purpose of the Cmicro Recorder has nothing to do with the purpose of the Cmicro Tracer. The user also has to use the Cmicro Tracer if he wants to get information about internal events. Additional information is then transmitted via the communications link and stored into a file which can be

displayed dynamically or later on. The performance of the communications link dictates whether real-time trace is possible. The more information traced via the link, the higher the performance should be.

The record mode must be switched on within the target (by using the C function interface on target), and on the host - see subsection "Record a Session" on page 3626).

Re-Playing an SDL Session

After producing a file with the record mode, the file can be replayed in order to run the SDL system in the same order through the same system states.

When the end of file is reached which is indicated with <Play mode off> to the target, the SDL system is able to react on ordinary environment signals coming from the real environment (if programmed by the user).

The play mode must be switched on within the target by using the SDL Target Tester Command Recorder-Play. No environment signals may be handled by the user in the C function `xInEnv` during the play.

Reaching the Erroneous System State

Sometimes, when a file is replayed, the erroneous situation can directly be viewed at another device, i.e. if the SDL system runs in an emulator with breakpoints set, or if there is, for instance, an LC display connected to the SDL environment.

In this case, it is not necessary to compare output files. In other cases, it might be of interest to do so as described later.

Comparing the Results

A comparison of two executions is possible, when two files containing the same type and amount of information are compared.

The procedure goes as follows:

- Change the format from binary into ASCII for both files (command "Convert-File" on page 3635).
- Use UNIX diff or an ASCII editor with options in order to evaluate the differences.

Commands for the Host and Debugging Facilities

The command interface and the debugging facilities are very helpful in finding errors produced within the target.

For instance, after an SDL interpretation error has occurred, it is possible to suspend the SDL system and to inspect the global state by viewing the signal queue(s). After the queue size has been re-dimensioned, the queue can be inspected again on the host site during a target execution.

There are several commands to check that timers are correctly processed.

The ability to set breakpoints during real-time execution and to execute a single step also helps to find an error situation.

The amount of trace information can dynamically be defined via commands. A differentiation is made between the information on the communications link and the information displayed.

Using the SDL Target Tester's Host

Introduction

The SDL Target Tester on host site is a tool chain consisting of the parts: GUI (sdtmtui), the logical part (sdtmt), the communications link gateway (sdtgate or sockgate) and the glue between these components the Cmicro Postmaster (sdtmpm). Sometimes, e.g. when the SDL Target Tester uses the MSC Editor, a further tool is executed called Cmicro Link (sdtmlnk).

The SDL Target Tester's host site is the tool, which allows communication with the target via a communications interface (see "The Communications Link's Host Site" on page 3718) to read or write to files (sdtmt), display information on the screen (sdtmtui) and/or the MSC Editor. It allows tracing of SDL and system events and has the ability to record and replay events to reproduce error situations.

Host features:

- Trace of SDL and system events
- Symbols and processes can be selected and filtered in order to reduce the amount of information
- Record and replay of SDL executions (error reproduction)
- Execution trace into file, to screen and to MSC Editor
- Files can be read in later and displayed either on screen or on the MSC Editor
- Small SDL Debugger

It is important to know that the host works independently from the target. The communication between both is built up automatically by the host if you start to communicate with the target. Thus, the host part of the SDL Target Tester remains the same executable when changing the target hardware! Only some adaptations in the configuration file `sdtmt.opt` have to be made, see "Preparing the Host to Target Communication" on page 3614.

The graphical user interface is described in "Graphical User Interface" on page 3656.

Different Ways of Using the SDL Target Tester

The standard method for using the SDL Target Tester is to connect a host with a target machine. A communications link implemented by the user must exist between host and target. The target can be seen as a remote system, where the SDL Target Tester running on the host machine can be considered a controlling unit. This is described in Figure 600.

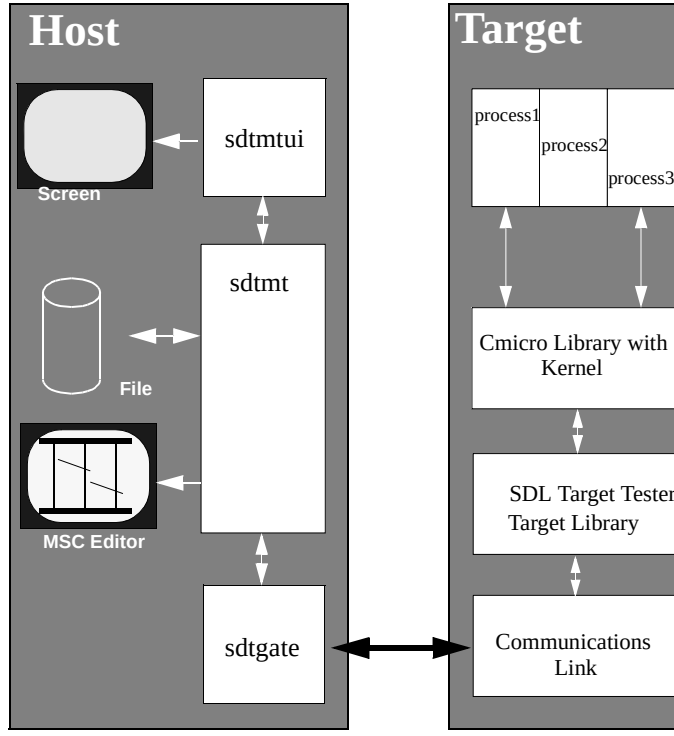


Figure 600: SDL Target Tester used for target control

The second way to use the SDL Target Tester is as a front-end for a host simulation. Therefore, a host simulation for the SDL System is built with the Targeting Expert. As described in Figure 601 the Gateway used for connection to the target is replaced by the host simulation.

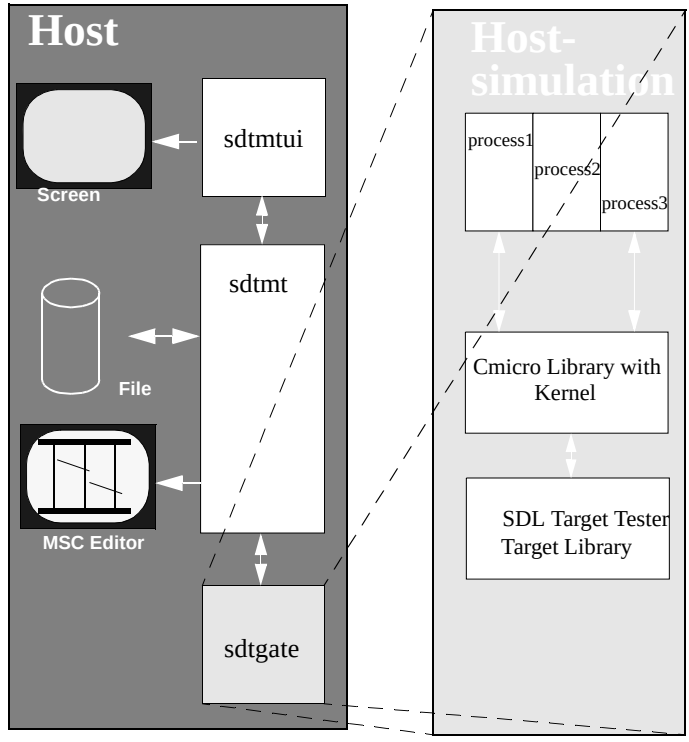


Figure 601: SDL Target Tester used for host simulation

Another way to use the host software is to use it purely as a conversion tool. This is of interest if a binary file has been written during earlier sessions with the target. The following picture shows the SDL Target Tester as a conversion tool:

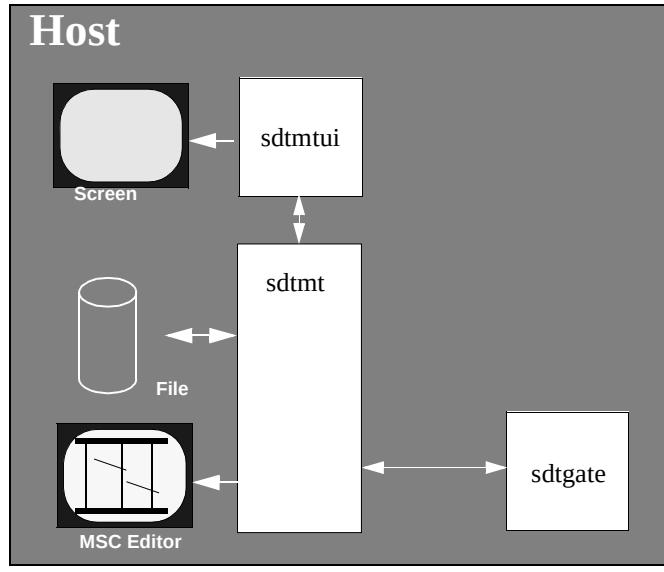


Figure 602: Sdtmt used for file conversion

Furthermore, by using the commands Output-NPAR and Output-PAR the SDL environment of the target or host simulation can be simulated with the host site.

Preparing the Host to Target Communication

After implementing the parts in the target, which are necessary to have a communication between host and target, the appropriate device on the host site is to be selected. This can be done in the file named

`sdtmt.opt`.

The Targeting Expert should be used to generate or modify `sdtmt.opt` (see "Configure the Host (Cmicro only)" on page 2948 in chapter 59, *The Targeting Expert*).

The Default Communications Link of Sdtmt

There is a V.24 interface implementation to transfer data between target and host. This is delivered as an additional executable called `sdtgate`.

The sdtgate default implementation uses the following protocol:

- XON/XOFF protocol possible
- 8 Bit data
- One start bit
- One stop bit
- No parity
- All data is transmitted as binary.
- Control characters are masked and de-masked.
- Binary protocol: Cmicro Protocol.

There is also a socket interface implementation to transfer data between target and host. This is delivered as an additional executable called `sockgate`.

The sockgate default implementation uses the following protocol:

- All data is transmitted as binary.
- Control characters are masked and de-masked.
- Binary protocol: Cmicro Protocol.

Communication Setup on the Host System

If sdtmt is invoked, it searches for the configuration file `sdtmt.opt`. This file is mandatory!

Each line beginning with a '#' is treated as a comment.

It contains information on target options such as:

- Target timer's scale and unit.
  These entries affect only the text trace.
- Target data types' endian, alignment and size

> **Caution!**
>
> The file needs to be updated if another target system is connected, or if you change from a host simulation to your real target.

The file `sdtmt.opt` should have the following entries in order to configure the communications link:

- USE_GATE:

  The path and the name of the used gate executable has to be specified here. In the delivered version of the SDL Target Tester, two executables called sdtgate and sockgate are included. They handle the V.24 and the socket interface on the host system.
  In case of a host simulation, the name of the executable which contains the simulated SDL system is inserted instead of the gateway.

- GATE_CHAR_PARAM_, GATE_INT_PARAM_

  When the gate is forked by sdtmt these values are handed over to the gateway to configure the interface.

  – V.24

    Select the device: `GATE_CHAR_PARAM_1 <devicename>`

    On UNIX `<devicename>` might be `/dev/ttya` or `/dev/ttyb` (Please ask your system administrator).

    In Windows `<devicename>` might be `COM1` or `COM2`

    Select the appropriate baud rate: `GATE_INT_PARAM_1 19200`

All other GATE_. . . entries are not used with V.24 implementation.

– Socket

Select the device: GATE_CHAR_PARAM_1 <IP-address>

<IP-address> has to be given in the standard format 192.168.1.4 or localhost.

Select the socket port number: GATE_INT_PARAM_1 12345

All other GATE_. . . entries are not used with Socket implementation.

- UNIT-NAME, UNIT-SCALE

  The target's timer scale is multiplied with the value of UNIT-SCALE and displayed in units UNIT_NAME in the SDL Target Tester. The default values are "sec" and "1".

- USE_AUTO_MCOD

  Switch on or of the use of the automatic message coder configuration on the host. Valid values are "yes" and "no".

  If the value is "no" you have to use the values described in the following to configure the message coder.

  If the value is "yes" the message coder will be configured by the target itself. Therefore it sends its configuration in a message at start-up.The following values will be ignored in that case.

Note:

The target has to be compiled with the flag XMK_USE_AUTO_MCOD defined

- LENGTH_

  The length (in octets) of the C basic types within the target must be given here. See the compiler manual for further information (header file limits.h).

- ALIGN_

  The alignment for each C basic type must be given here.

  Alignment is the distance (counted in bits) from the first bit of a character to the named basic type. The Character is stored at an address dividable by four. See Figure 603.

  Allowed values are 8, 16, 32

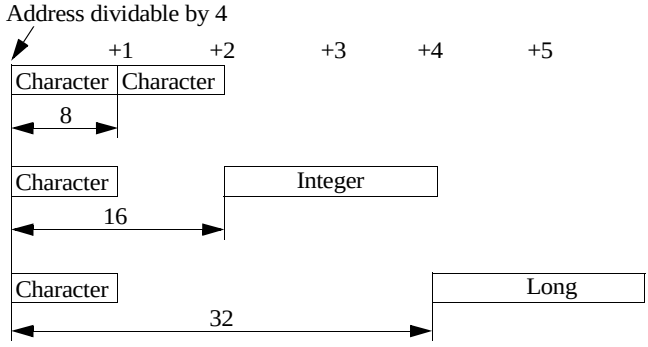


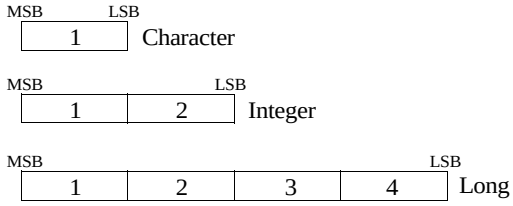
Figure 603: Target's alignment types

- ENDIAN_

  The order of the C basic types within the target's memory must be given here. See the microcontroller data sheets and Figure 604 to get further information.

  Allowed values are: 1, 12, 21, 14, 41, 23, 32, 18, 81

The octets of some C basic types are numbered as follows:



According to the type sizes above, the following Endians are possible within the target's memory:

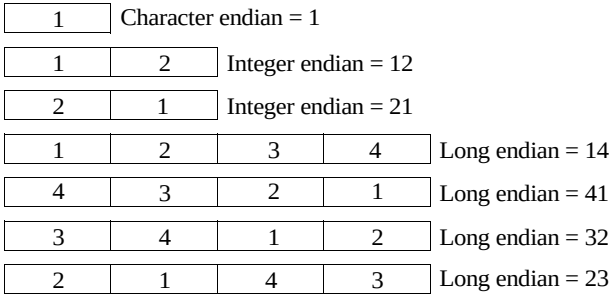


Figure 604: Definitions of the target's endian

Example 614: sdtmt.opt ––––––––––––––––––––––––––––––––––––

```
USE_GATE      $(sdtdir)/sdtgate   (on UNIX)
USE_GATE      $(sdtdir)\sdtgate.exe (in Windows)
GATE_CHAR_PARAM_1     COM2
GATE_CHAR_PARAM_2     0
GATE_CHAR_PARAM_3     0
GATE_INT_PARAM_1      9600
GATE_INT_PARAM_2      0
GATE_INT_PARAM_3      0

UNIT-NAME             sec
UNIT-SCALE            0.001

USE_AUTO_MCOD         no

LENGTH_CHAR           1
LENGTH_SHORT          2
LENGTH_INT            4
LENGTH_LONG           4
LENGTH_POINTER        4
LENGTH_FLOAT          4
LENGTH_DOUBLE         8
ALIGN_CHAR            8
```

```
ALIGN_SHORT          16
ALIGN_INT            32
ALIGN_LONG           32
ALIGN_POINTER        32
ALIGN_FLOAT          32
ALIGN_DOUBLE         32
ENDIAN_CHAR           1
ENDIAN_SHORT         12
ENDIAN_INT           14
ENDIAN_LONG          14
ENDIAN_POINTER       14
ENDIAN_FLOAT         14
ENDIAN_DOUBLE        18
```

Invoking the SDL Target Tester's Host

Command Line Options of the SDL Target Tester

Option	Description
`-h`	Show a short help on the command line options.
`-v`	Show the version of the SDL Target Tester without opening it.
`-t <targetdir>`	Change the working directory to <targetdir>.

Invocation from the Organizer

There are two ways to invoke the SDL Target Tester from the Organizer with the actual project.
First you can use the menu *Tools > SDL >Target Tester UI*.
The other way is to add the following lines into the file `org-menus.ini`. For more information, see *chapter 11, The Public Interface*.

```
[MENU]
Name=&Cmicro
[MENUITEM]
ItemName=&Tester
Separator=0
StatusbarText=Start Target Tester
ProprietaryKey=1
AttributeKey=0
Scope=ALWAYS
ConfirmText=
ActionInterpretation=OS_COMMAND
BlockCommand=0
FormattedCommand=sdtmtui -t%v
```

Note:

Since code generation and compilation for Cmicro is only possible with the Targeting Expert, and the Organizer does not know about the directory where the Targeting Expert generates all files, you will be prompted automatically to enter the right directory.

In host simulation mode, internally, the -s flag is used additionally. The output of the Tester then contains some more helpful information that cannot be given in target debug mode.

Invocation from the Command Line or Desktop

Before using the SDL Target Tester described in the sections below, it is assumed, that a communications link between host and target exists (see "Connection of Host and Target" on page 3688), that the target executable has included the SDL Target Tester and that the trace is switched on within the target.

Several files have to be read from the user's current working directory.

On UNIX, the SDL Target Tester's host simply needs to be started from the working (project) directory.

1. Change the current directory to the project directory (assuming that this directory is `~/cmicro_project`).
   `cd ~/cmicro_project`
2. Now, type
   `sdtmtui`

Note:

If the command `sdtmtui` is not found, the `$path` variable needs to be set up correctly. The user should ask his system administrator or the person that is responsible for the SDL Suite environment.

In Windows it is recommended that you create a new shortcut to the SDL Target Tester in the Window's Start menu or on the desktop. The shortcut must contain the following entries in its properties sheet:

- Target:
  `<instdir>\bin\wini386\sdtmtui.exe -t<targetdirectory> -`

- Entry Start in:
  `<instdir>\bin\wini386\`

Note:

The user should ask his system administrator about adding new links to the Start menu or the desktop.

The SDL Target Tester's host site is started when you double-click the SDL Target Tester icon.

The SDL Target Tester's host tool chain responds by showing the SDL Target Tester window:

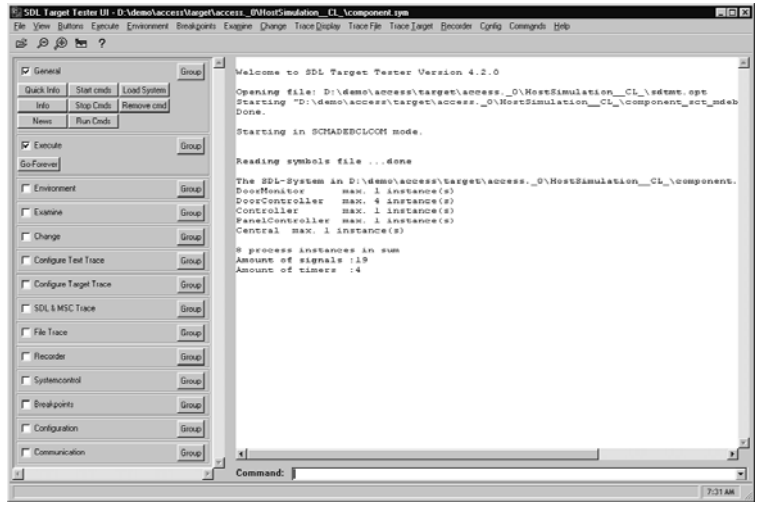


Figure 605: SDL Target Tester window

In the text area of the UI the following is displayed at startup:

```
Welcome to SDL Target Tester Version 6.3.0

Opening file: D:\...\sdtmt.opt
Starting "D:\...\component_sct_mdebcom.exe" ...
Done.

Starting in SCMADEBCOM mode.
```

```
Reading symbols file ...done

The SDL-System in D:\...\component.sym contains these processes:
DoorMonitor max. 1 instance(s)
DoorController max. 4 instance(s)
Controller max. 1 instance(s)
PanelController max. 1 instance(s)
Central  max. 1 instance(s)

8 process instances in sum
Amount of signals :19
Amount of timers  :4
```

After the target is reset its whole configuration is shown:

Example 615: Target's Start-up / Configuration Message ----------

```
The current target configuration is:

Compiled Cmicro KERNEL Features:
  Preemption is                          not active
  Signal Priorities are                  used
  Receiver PId in signals are            used
  Sender PId in signals are              used
  Sending Signals the simple way is      not used
  Pid addressing is                      possible
  SDL System Stop can                    not be used
  Additional errorchecks are switched    ON
  Timer handling is                      implemented
  Timers with parameters are             not used
  Dynamic create is                      implemented
  Save is                                not implemented
  Dynamic stop is                        not implemented
  Only single instance processes         NO

Compiled SDL Target Tester Features:
  Host commands can                      be used
  Profiling can                          be used
  Debugging can                          be used
  Trace is                               implemented
  Signalfilter can                       be set
  Time stamps are                        implemented
  Recorder can                           not be used
  Realtime play can                      not be used
  Trace is currently switched            ON
  Auto MessageCoder is                   ON

Selected TARGET Modes:
  Recorder is currently switched         OFF
  Player is currently switched           OFF

Message Coder configuration: (size in octetts)
-------------------------------------------------------
  SDL process ID's (xPID)               :  4
  SDL state ID's   (xmk_T_STATE)        :  1
  SDL signal ID's  (xmk_T_SIGNAL)       :  1
  SDL time values  (xmk_T_TIME)         :  4
  Message length  (xmk_T_MESS_LENGTH )  :  1
-------------------------------------------------------

SDL Data Type sizes (octetts):
-------------------------------------------------------
  Boolean    :  1 | Bit        :  1 | Character  :  1
  Charstring :Ptr | Integer    :  4 | Natural    :  4
  Real       :  8 | PId        :  4 | Duration   :  4
  Time       :  4 | Null       :  4 | Octet      :  1
  Octetstring: 12 | ObjId      : 16
-------------------------------------------------------

  Target was generated at: Mon Jan 01 12:00:00 2001

SDL System ready for configuration.

Start with "Go-Forever".
```

Note:

Within the target's configuration message the timestamp (seconds since the 1st of January 1970) of the generated C file is sent to the host, too.

If this timestamp is not equal to the one in the symbols file (`.sym`) a message box is displayed giving hints as to what has to be done.

Getting a Target Trace

After the start-up message is received the target is ready to be traced.

The command `Go-Forever` starts the SDL system in the target and the static create trace messages will be sent.

When a process instance executes actions within a transition, trace information describing the different actions is either printed on the screen, on the MSC Editor or written into a file specified by the user. The amount of information printed can be selected using the trace commands in the monitor system. All trace information is self explanatory. This trace example was created from the AccessControl:

Example 616: Trace for the AccessControl ––––––––––––––––––––

```
*** STATIC CREATE
*     PId       : DoorMonitor:0
*   TASK  LastOffspring :=
*** NEXTSTATE Idle

*** STATIC CREATE
*     PId       : Controller:0
*   TASK  NextDoor :=
*   OUTPUT of  Allocate to DoorMonitor:0
*     Prio      : 100
*     Now       : 0.000000 sec
*     Param(s)  : 1
*   SET  MyTimer
*     Time      : 10.000000 sec
*     Value     : 0
*** NEXTSTATE WaitAllocated

*** STATIC CREATE
*     PId       : PanelController:0
*   OUTPUT of  Display to ENVIRONMENT
*     Prio      : 100
*     Now       : 0.000000 sec
*     Param(s)  : 1
*** NEXTSTATE Idle
```

```
*** STATIC CREATE
*     PId      : Central:0
*   TASK  NextFree :=
*** NEXTSTATE Idle
```

Debugging – A Few Guidelines

The following few guidelines may help to understand the different possibilities when searching for a specific command.

Breakpoints

A lot of error situations may occur when the system is downloaded for the first time. Possibly the basic functions needed to allow the SDL system to run the target do not work. Possibly the compiler adaptations are wrong. The initialization of the environment may be incomplete. The compiler may generate faulty object code. The memory of the target may be exceeded, i.e. either variables, constants, program code, stack or dynamic allocated memory. Also, real-time requirements may not be fulfilled.

Breakpoints can greatly help to discover such situations. The breakpoint logic is especially useful as it has only a negligible impact on the systems real-time performance.

Setting Breakpoints

Does a specific process instance (pid) receive a specific signal?

```
BPI processname nr signalname
```

Does a specific process instance (pid) go into a specific state (does it end any transition, which leads to that state)?

```
BPS processname nr statename
```

Trace Scaling

The target's trace can be scaled. Sometimes it is necessary to omit trace altogether.

This may be necessary to reduce the amount of trace messages via the communications link, but can also be used to expand the view of some details.

Switch off the trace of signal parameters

```
Tr-Params 0
```

or switch it on again.

```
Tr-Params 1
```

The main interest may be lay on the work of one special process. This process can be explicitly defined with

```
Tr-Process processname nr
```

The trace for all other processes will be switched off.

Corresponding to this command the scope view can be set to a specific signal

```
Tr-Signal signalname
```

If the trace of the whole SDL system should be reduced, e.g. the trace of TASKs and DECISIONs is of no interest

```
Tr-Detail 4
```

switches the trace of these symbols off.

The command Tr-Detail can be used with five levels, for further information see "Tr-Detail" on page 3652.

Record a Session

> **Note:**
>
> The SDL Target Tester's Record and Play functions are only available if a Cmicro Recorder license is available.

See "Invoking the SDL Target Tester's Host" on page 3619 to start-up the SDL Target Tester. When the target's start-up message is displayed in the text area, the target must be configured to record a session, i.e. the Recorder must be switched on in the target.

If the user wants to replay a recorded session in real-time mode he must be sure that the target executable is compiled with signals including time stamps (see "XMK_USE_SIGNAL_TIME_STAMP" on page 3508).

The user has to do several things

1. Open an output file by using the command
   `Output-File filename`
2. Switch the Cmicro Recorder on by using
   `Recorder-On`
3. Start the SDL system with
   `Go-Forever`

As with a standard trace the target's messages are displayed in the text area. The user can now use the target as if the SDL Target Tester is not involved. All traces and recorded messages are stored into the output file.

4. To end the session the Cmicro Recorder should be stopped:
   Recorder-Off
5. And the output file has to be closed:
   Close-File

Re-Play a Recorded Session

Before replaying a session it is required to record a session by following the steps listed in "Record a Session" on page 3626.

First the target has to be reset. The start-up message reappears.

Now the user can start the replay with the commands

1. `Input-File filename`
   where `filename` is the name of the file written during a recording session (this file must contain Cmicro Tracer messages as well as Cmicro Recorder information).
2. Switch the play mode on:
   `Recorder-Play`
3. If desired switch the real-time play mode on:
   `Recorder-Realtime`
   Remember: the target has to be compiled using this feature.
4. Start the target:
   `Go-Forever`

The target now starts by tracing the static creates and in accordance with the design of the SDL system the first transitions are executed and traced.

Now the SDL Target Tester's host inserts all the signals coming from the environment so that the recorded session will be replayed in the same order.

Note:

The real environment is not being polled during the replay session.

Restrictions of the SDL Target Tester

The SDL Target Tester has a few restrictions:

- Only values of the following SDL data types are displayed in readable form: Bit, Boolean, Char, Charstring, Integer, Natural, Octet, Real, PId, Duration and Time.
  Syntypes, Structures, Enumeration Types and Arrays which contain only SDL-data types can be decoded correctly.
- Complex data types, e.g. Array of Structures indexed by syntype may be decoded wrong due to alignment.
- If the SDL Target Tester encounters a unknown data type, decoding is interrupted. The first unknown data type is printed to the screen and the rest of the parameters is displayed as a hexadecimal buffer.
- Record and play mode may not be entered together.
- During the play mode only certain commands are allowed. The available commands are marked with an asterisk.
- Any SDL sort that is implemented as pointer in C (like Charstring, Object_Identifier, Ref, Own, etc.), cannot be handled in trace, record or play mode. This comes as a result of the implementation of trace (explained in chapter 66, *The Cmicro Library*). The trace does not distinguish if there is a buffer containing a pointer to be traced or not, because there is no symbol information generated into the target system for efficiency.

SDL Target Tester Commands

Syntax of SDL Target Tester Commands

Command Names

Command names are entered character by character on the keyboard. Each command consists of ASCII characters, terminated by a carriage return. A command is interpreted, after the carriage return is given. Commands may have parameters. Command names may be abbreviated by giving sufficient characters to distinguish it from other command names. The command names are interpreted from left to right. There is no distinction between upper and lower case letters.

Consider, as an example, the command ?Breaklist. The command may be entered as “?b”. However, if only ‘?’ is typed, sdtmt will respond with the error message:

```
Command was ambigous, choose one from:
< list of commands >
```

since the command cannot be distinguished from, for example, the ?Queue command. As additional information a list of all commands which would fit to the entered command and which are allowed in the current context is given.

Command Parameters

A parameter is separated from the command name by one or more spaces. The same is applicable to the separation of two parameters. There is no distinction between upper and lower case letters.

Each parameter is mandatory from a syntactical point of view and must be specified. No parameters may be left out.

If the parameter list following a command name is not complete, a dialog window is opened to enter the missing parameter(s).

Specifying more parameters than the command can handle, sdtmt prompts

```
**ERR:Extra parameters specified in command.
```

If the type of any parameter, which was entered, does not match the expected one, then sdtmt responds with an appropriate error message according to the expected type. Type

```
help commandname
```

or refer to the manual.

Abbreviation of SDL Names

Command parameters that are SDL names, may also be abbreviated, as long as the abbreviation is unique. If a name is equal to the first part of another name, as for example Wait and Wait_For_Me, then any abbreviation of Wait is ambiguous. However, if all characters of the shorter name are given (Wait in this example), this is considered as an exact match and the short name will be selected.

Errors in Commands

If an error is detected in a command name or in one of its parameters, an appropriate, self explanatory error message is displayed on the screen. The command will be ignored.

Input and Output of Data Types

<Processtype Instance>

Process type is an ASCII string identifying the SDL name of the process. Process names must be unique for all processes within the system. It is also possible to use the decimal value of the process type. The decimal value can be found by looking into the symbol file, which is also automatically generated.

The separator for process type and instance is a blank.

Instance

The value of the instance number is a decimal value beginning from 0 for the first instance of a process type. If the SDL System contains only (x, 1) declarations on SDL level, then the user has to specify “0” in any case.

Note:

There is a difference between the notation in the simulator and in the SDL Target Tester concerning instance numbering. The simulator always uses the value one as the first instance number whereas the SDL Target Tester begins numbering these at zero!

Caution!

If no symbol file was specified, then neither process type nor instance is checked within sdtmt! If the target is compiled without error checks, that might lead to a fatal error in the target system.

Symbols

Symbols are read in from a file called `<systemname>.sym`. There are a lot of error checks within the SDL Target Tester's host site which only work if a symbol file has been loaded.

Help

SDL Target Tester gives the user some quick help when entering:

```
help
```

A complete list of commands will be displayed on the screen. Commands, which are available in the current mode, are marked with an asterisk. If a command is entered which is not available in the current context the SDL Target Tester responds with:

```
Sorry. The current mode of sdtmt does not allow you
to enter that command.
The command has been ignored.
```

To get a short explanation, type:

```
help <commandname>
```

for instance type:

```
help help
```

To get a complete list of commands, type:

```
help *
```

Most of the commands marked with the type “Remote” can only be executed (in the target) if the flag XMK_ADD_MICRO_COMMAND is defined in the file `ml_mcf.h`. See “Manual Scaling” on page 3486.

??

Parameters:

```
(None)
```

Type: Local

Open a dialog with a list of all allowed commands. By pressing the OK-Button the selected command is copied to the Input line.

Active-Timer

Parameters:

```
<Processtype Instance> <Timername> <Timerparameter>
```

Type: Remote

Check, if a specific timer is set by a process, and has not been expired.

> **Note:**
>
> The input of a timer parameter is mandatory. The type of the parameter has to be an integer value. For a timer without parameter enter a 0 in the parameter dialog.

?All-Processes

Parameters:

```
(None)
```

Type: Remote

> **Note:**
>
> This command is obsolete. Use ?Process-State * * instead.

BA

Parameters:

```
(None)
```

Type: Remote

This command requests to clear all the breakpoints in the break list.

BC

Parameters:

```
<breakpoint number>
```

Type: Remote

This command requests to clear the given breakpoint number from the breakpoint list. The breakpoint number depends on the total number of defined breakpoints.

BP

Parameters:

```
<Processtype Instance> <SignalName> <StateName>
```

Type: Remote

This command sets a breakpoint on the input and the state of a process. Execution is halted only if the given input in the given state is detected.

No parameter is optional. These combinations are allowed:

```
processname nr * statename     ->Use command BPS
processname nr signalname *    ->Use command BPI
processname nr * *
processname nr signalname statename
```

BPI

Parameters:

```
<Processtype Instance> <SignalName>
```

Type: Remote

This command sets a breakpoint on the input of a given process. Execution is halted on any input of the signal in this process.

It is not mandatory to provide a parameter. These combinations are allowed:

```
processname nr *
processname nr signalname
processname *
processname signalname
```

BPS

Parameters:

```
<Processtype Instance> <StateName>
```

Type: Remote

This command sets a breakpoint on the state of a given process. Execution is halted if the given state is reached.

No parameter is optional. These combinations are allowed:

```
processname nr statename
processname *
processname statename
processname nr *
```

?Breaklist

Parameters:

```
(None)
```

Type: Remote

This command requests the current breaklist from the target and displays it on the screen as it was entered. If a symbol file has been specified during the invocation of sdtmt, then process IDs, signal IDs and state IDs are displayed with their symbolic value.

Change-Directory

Parameters:

```
<directoryname>
```

Type: Local

The working directory of the current SDL Target Tester session is switched to `<directoryname>`.

Close-File

Parameters:

```
<filename>
```

Type: Local

The current output/input file (including binary trace) with the name `<filename>` will be closed when using this command. (See “Output-File” on page 3640 and “Input-File” on page 3639)

?Coder

Parameters:

```
(None)
```

Type: Local

After invoking this command the current settings of the Target Message Coder TMCOD are display in the text area. The settings given here have to be previously made in the configuration file `sdtmt.opt` or have been sent by the target itself. See "Preparing the Host to Target Communication" on page 3614.

Continue

Parameters:

```
(None)
```

Type: Remote

This command can be used to continue processing, i.e. after a breakpoint was h. Notification is given when the command cannot be processed within the target system.

Convert-File

Parameters:

```
<input filename> <output filename>
```

Type: Local

This command converts the trace messages of the target (stored in the `<input filename>`) into readable ASCII trace. This ASCII trace is stored into the `<output filename>`.

Create

Parameters:

```
<Parent-Processtype Instance> <Child-Processtype>
```

Type: Remote

This command creates an SDL process of the given type `<Child-Processtype>`. The creating process must be specified in `<Parent-Processtype Instance>`. If there is a free process instance, which is currently dormant, it will be created and the start transition will be executed. This can be seen in the trace.

If the system is under a break condition, the command will be ignored as it is not possible to send signals (dynamic creation uses signal mechanism).

> **Note:**
>
> Please specify the `<Parent-Processtype Instance>` according to the SDL system. There is no check if the parent is specified correctly from the semantic point of view.

Disable-Timer

Parameters:

`(None)`

Type: Remote

This command disables the processing of timers.

The command affects the SDL timers only! However, timers which are already expired and are in the queue will be input as normal. Keep in mind that the disabling of timers changes the system behavior.

Display-Off

Parameters:

`(None)`

Type: Local

The ASCII trace into the text area is switched off.

Display-On

Parameters:

`(None)`

Type: Local

The ASCII trace into the text area is switched on.

Enable-Timer

Parameters:

`(None)`

Type: Remote

This command enables the processing of timers again, if it was previously disabled with the command Disable-Timer. Due to the fact that time (and so the system time) cannot be stopped, it might happen that

all the timers set earlier may expire at the same time this command is entered. This may cause illegal system behavior, i.e. a queue overflow and the call of the ErrorHandler or another illegal behavior.

?Errors

Parameters:

`<const>`

Type: Remote

Query last occurred error. The target system traces the last error within the SDL execution and stores it. The result will be displayed on the screen, both as a decimal value and as ASCII text with an explanation of the error situation.

If the value `<const>` is not equal to zero, further information about the error is given from this manual (the on-line Help Viewer is started).

Especially, when the SDL system is executed for the first time in a new hardware environment, the command may help in finding problems. It is also useful after a long execution of the SDL system, as any error situation arising may be detected. All the errors, warnings and information, which are handled by the ErrorHandler are explained in the section "User Defined Actions for System Errors – the ErrorHandler" on page 3548 in chapter 66, *The Cmicro Library*.

Exit-Single-Step

Parameters:

`(None)`

Type: Local

The single step mode (which has been entered with Single-Step) is finished.

Get-Configuration

Parameters:

`(None)`

Type: Remote

The target responds with its configuration. Compiled features are displayed in the text area as well as the current settings (e.g. trace on/off).

Go-Forever

Parameters:

```
(None)
```

Type: Remote

After initializing the target (for example switching the recorder mode on) the target is started with this command. It is necessary that the target is compiled with the flag XMK_WAIT_ON_HOST.

Help

Parameters:

```
[<commandname>  or  *]
```

Type: Local

A complete list of commands will be displayed on the screen. Commands available in the current mode of the SDL Target Tester are marked with an asterisk.

Example 617 ----------

Type

```
Command>help <commandname> ,
```

to get a short explanation of one or several commands, e.g.

```
help help
```

which is the same as

```
he he
```

Example 618 ----------

A list of all commands beginning with **b** can be requested by typing:

```
help b
```

A complete list of commands can be requested by typing:

```
help *
```

Input-File

Parameters:

`<filename>`

Type: Local

The file named `<filename>` is opened to get the stored trace information (binary format). This command must be used if, for example, a recorded session should be replayed. The file `<filename>` must contain trace information in binary format.

Line

Parameters:

`(None)`

Type: Local

The command displays the line status of the communications interface. It can be used for checking if characters, frames and XONs / XOFFs have been received/transmitted and the kind of device that is actually connected.

?Memory

Parameters:

`(None)`

Type: Local

If the Cmicro Memory Management is selected (the flag XMK_USE_SDL_MEM must be defined, please view "Use of Memory" on page 3515 in chapter 66, *The Cmicro Library*) this command offers the advantage of checking currently allocated memory.

News

Parameters:

`(None)`

Type: Local

The SDL Target Tester Release's news are displayed in the text area.

Next-Step

Parameters:

`(None)`

Type: Remote

This command performs one step (one transition) if single step is active. The Single-Step command must be given prior to this.

Nextstate

Parameters:

```
<Processtype Instance> <Statename>
```

Type: Remote

This command sets the state of the given process instance to the value of `<Statename>`. `<Statename>` must be one of the states which are defined for the corresponding process type in SDL. Normally, that only makes sense if the system is under break condition or is idle.

Output-File

Parameters:

```
<filename>
```

Type: Local

All the target's trace information is stored as binary data into the file `<filename>`. This command can be used to store the information of a recording session in binary format.

Output-NPAR

Parameters:

```
<Receiver-Processtype Instance> <Signalname>
```

Type: Remote

An Output from the environment without any parameters is sent into the SDL system.

Output-PAR

Parameters:

```
<Receiver-Processtype Instance> <Signalname> <Signal
Parameter>
```

Type: Remote

An output from the environment into the SDL system with a signal including parameters is sent into the SDL system. The `<Signal Parameter>` has to be given as a buffer of hexadecimal values. See Example 619.

Example 619: Output with parameters ––––––––––––––––––––––––

```
Output-PAR Process_A 0 Signal_1 "00-12-a3"
```

Signal 'Signal_1' is sent to the instance 0 of process type 'Process_A'. 'Signal_1' contains parameters with the length of three octets and the values 0x00, 0x12 and oxa3. The ordering of the parameter octets must fit into the signal's parameter structure and the target's memory layout (alignment, endian etc.).

––

Options-File

Parameters:

`<filename>`

Type: Local

The options file `<filename>` is read in so that the message coder can be initialized. By default the options file `sdtmt.opt` (see "Preparing the Host to Target Communication" on page 3614) is read in when the SDL Target Tester is started.

Page-File

Parameters:

`<filename> <EventsPerPage>`

Type: Local

The file `<filename>` containing trace information (in binary format) is read in and displayed as ASCII trace in the text area. The constant decimal value `<EventsPerPage>` gives the amount of trace information to be displayed until the user confirms the next page.

Print-Conf

Parameters:

`(None)`

Type: Local

This command gives all configuration information together, see the commands ?Coder and Line.

?Process-Profile

Parameters:

```
<Processtype Instance>
```

Type: Remote

The profiling of one process instance can be requested with this command. The result is the transition execution time of the longest transition and the last transition of the specified process instance. The flag XMK_ADD_PROFILE must be set when compiling the target executable to have access to this command in the target.

Please view the section "Initialization" on page 3504 in chapter 66, *The Cmicro Library*.

?Process-State

Parameters:

```
<Processtype Instance>
```

Type: Remote

Query the state of the given process(es).

The amount of queried process instances can be configured:

- If you specify a valid process instance (?Process-State P1 0), the current state of this instance is printed.
- If you enter a process type and * (?Process-State P1 *) you get the states of all instances of this type.
- If you enter ?Process-State * * all possible Process instances of the system are queried.

Depending on the traffic load on the communications interface the responses may be delayed.

Caution!

The results printed on the screen may be inconsistent. That depends on what the SDL system is doing during the query procedure. The results will be correct, however, if the whole SDL system is idle.

?Queue

Parameters:

(None)

Type: Remote

Some characteristics of the queue will be displayed. The command is very useful for determining the queue's dimensions. There is a peak hold, which shows the maximum number of entries in the queue since the system's start.

Example 620 ----------

After typing the command, information is printed on the screen, which may look like:

```
Current Queue state:
Queue size dimensioned to:Max.20 entries
Peak hold                 :1 entry/entries
Currently                 :0 entry/entries in queue
Queue memory located at   :36a40010,x
```

The explanation for this is: a maximum of 20 entries in the queue can be handled by the system. Since system start, there was never more than one entry being used. Currently, no signal instance has been detected in any process input port. The last value displays the physical memory allocation of the queue, which helps in debugging.

Recorder-Delay

Parameters:

<duration>

Type: Local

If a recorded session should be replayed it is possible to insert each environment signal with a delay of <duration> seconds. This feature can be used instead of the real-time play (Recorder-Realtime).

Recorder-Off

Parameters:

(None)

Type: Remote

Switches the Cmicro Recorder off. This command may be specified any time during a recording session. It can be used in order to reduce the amount of information sent via the communication interface.

Recorder-On

Parameters:

(None)

Type: Remote

Switch Cmicro Recorder to record mode. It is not a good idea to enter this command when the SDL system is not idle. (Furthermore, the question is, whether it makes sense to start a record session in the middle of an SDL execution or from the start-up of the SDL system.) It may not be relevant to start a recording session after the system has started.

> **Note:**
>
> The SDL Target Tester's Record and Play functions are only available if a Cmicro Recorder license is available.

Recorder-Play

Parameters:

(None)

Type: Remote

The Cmicro Recorder's play mode will be set. This command can be entered in a few states of the Cmicro Kernel only. Notification is given if necessary. The same explanation as for Recorder-on apply.

> **Note:**
>
> The SDL Target Tester's Record and Play functions are only available if a Cmicro Recorder license is available.

Recorder-Realtime

Parameters:

(None)

Type: Local

The play mode of the SDL Target Tester is started as real-time play. It is necessary to compile the target with signals including time stamps.

The SDL Target Tester's host sends all environment signals with the recorded time stamp into the target while replaying. Without using this

command all signals are inserted with time 0 which leads to an immediate execution in the target.

Reinitialize

Parameters:

```
(None)
```

Type: Remote

This command tries to reinitialize the system. The reinitialization may fail, if the hardware drivers cannot be reinitialized again. Furthermore, all global auto variables in the user's C program parts cannot be reinitialized. However, if there is a clean implementation of driver initialization and de-initialization, and initialization of global variables, the command will work well.

Remove-All-Signals

Parameters:

```
<Processtype Instance>
```

Type: Remote

Remove all signals of the given receiver from the signal queue.

Remove-Command

Parameters:

```
(None)
```

Type: Local

For each command, which is sent to the target, the host waits for an explicit acknowledgment. The last command can be removed, so that the host does not wait for that acknowledgment. That can be used when the communication is hanging for any reason, for example, when the command could not be sent to the target, or was not received by the target, or the acknowledgment could not be received by the host.

Remove-Queue

Parameters:

```
(None)
```

Type: Remote

All signal instances in the queue will be removed. The signal queue will be empty.

Remove-Signal

Parameters:

```
<Processtype Instance> <Signalname>
```

Type: Remote

Remove a specific signal of the given receiver from the queue.

> **Note:**
>
> E.g. when a breakpoint is hit the signal which led to the current transition still remains in the signal queue. This means the first signal in the queue cannot be removed.

Reset-All-Timers

Parameters:

```
<Processtype Instance>
```

Type: Remote

This command resets all SDL timers of the given process instance.

Reset-Timer

Parameters:

```
<Processtype> <Instance> <Timername>
<Timerparameter>
```

Type: Remote

This command resets a specific SDL timer of the given process instance and with a specific timer value.

> **Note:**
>
> The input of a timer parameter is mandatory. The type of the parameter has to be an integer value. For a timer without parameter enter a 0 in the parameter dialog.

Resume

Parameters:

```
(None)
```

Type: Remote

Resume Cmicro Kernel, after a suspend has been accepted. The same explanations as for the Suspend command apply.

Run-Cmd-Log

Parameters:

`<filename>`

Type: Local

The initialization commands in the log file `<filename>` will be executed.

> **Caution!**
>
> In principle it is possible to use all kinds of commands in this command log file. But the correct execution cannot be guaranteed because there are to many interactions between the SDL Target Tester, the Gateway (`sdtgate.exe`), the target and the Organizer.
>
> E.g. the MSC Editor can only be started from a command log file if the requested performance of the host machine is supported.

> **Caution!**
>
> It is still possible to use debugging commands in this log file, too. But the user has to think about the usefulness as all commands will be executed directly after one another without getting the correct timestamp of execution.

Scale-Timers

Parameters:

`<factor>`

Type: Remote

Scale time when setting a timer.

Often there is a need to simulate time during an SDL execution. By specifying this command, it is possible to make SDL timers run faster or slower. The command is applied on newly set timers only, i.e. if a timer is already running then it remains unaffected from the time scale. Due to this fact, the command should only be given at the start of the SDL system.

In record mode the scaled time will be stored. This results in time being scaled in play mode. It is impossible to scale time manually in play mode!

Set-Timer

Parameters:

```
<Processtype Instance> <Time value> <Timername>
<Timerparameter>
```

Type: Remote

Set an SDL Timer for the given process instance with the given time value in units. The time value is a float value re-calculated to the target's timer ticks. Please view "UNIT-NAME, UNIT-SCALE" on page 3616.

> **Note:**
>
> The input of a timer parameter is mandatory. The type of the parameter has to be an integer value. For a timer without parameter enter a 0 in the parameter dialog.

Example 621 Setting a timer without parameters

```
set   <Processtype> <Instance> 4711 < timername> 0

set   <Processtype> <Instance> 1.456 <timername> 0
```

Shutdown

Parameters:

```
(None)
```

Type: Remote

The command tries to shutdown the system, which means normally to exit the main program. The user can define the actions on `exit` within the target. The same explanations as for the reinitialization command apply.

Single-Step

Parameters:

```
(None)
```

Type: Remote

This command switches the single step mode on. Either Next-Step or Continue may follow.

Note:

Timers are checked normally during single step and may expire, as documented for the Disable-Timer and Enable-Timer commands. It may be better to disable the timers (with Disable-Timer) before going into single step.

Start-Cmd-Log

Parameters:

`<filename>`

Type: Local

All the initialization commands will be written to the log file `<filename>`. Please view the command "Run-Cmd-Log" on page 3647, too.

Start-Gateway

Parameters:

`(None)`

Type: Local

This command is used to start the communication in the default implementation of the sdtgate or a host simulation.

Start-MSC-Log

Parameters:

`<level> <"Diagramname">`

Type: Local

The MSC Editor is started and the target's trace messages will be displayed in the MSC Editor. The `<level>` specifies the amount of MSC symbols:

0 - Basic Message Sequence Chart
1 - like level 0 but with the trace of states
2 - like level 1 plus the trace of actions
4 - Basic Message Sequence Chart to file
5 - like level 0 but with the trace of states to file
6 - like level 1 plus the trace of actions to file

Start-SDLE-Trace

Parameters:

```
<Processtype>  or
*
```

Type: Remote

The trace on SDL symbol level is started for the specified process. The Organizer has to be running and the target needs to be compiled with the flag XMK_ADD_SDLE_TRACE (see "Trace" on page 3506 in chapter 66, *The Cmicro Library*).

Start-Trace-Log

Parameters:

```
<filename>
```

Type: Local

A log file containing the whole ASCII trace of the SDL Target Tester will be written.

Stop

Parameters:

```
<Processtype Instance>
```

Type: Remote

This command stops an SDL Process according to the SDL semantics, no matter if the system is suspended or under break condition. The command is, after it is received, directly executed within the target. All signals which are in the queue for this process are removed, including create signals.

Stop-Cmd-Log

Parameters:

```
<None>
```

Type: Local

The write of initialization commands will be stopped. Please view the command "Run-Cmd-Log" on page 3647, too.

Stop-Gateway

Parameters:

```
(None)
```

Type: Local

This command is used to stop the XON communication in the default implementation of the sdtgate.

Stop-MSC-Log

Parameters:

(None)

Type: Local

The trace on the MSC Editor is stopped.

Stop-SDLE-Trace

Parameters:

<None>

Type: Local

The trace on SDL symbol level is stopped.

Stop-Trace-Log

Parameters:

<None>

Type: Local

The trace to the trace log file is stopped. The file is closed.

Suspend

Parameters:

(None)

Type: Remote

Disables the Cmicro Kernel from scheduling. The transition currently running will not be affected and will be ended first, before the command is accepted. The suspended SDL system cannot process any signals, neither internal signals nor signals coming from the environment. This may lead to a queue overflow. It is recommended to disable the timers to prevent this.

System

Parameters:

<systemname>.sym

Type: Local

The automatically generated symbol file `<systemname>.sym` is loaded. The symbolic names for process types and Signal IDs can only be used if this file is read in.

Additional error checks can be performed within the SDL Target Tester's host if this file is read in.

?Timer-Table

Parameters:

`(None)`

Type: Remote

Some characteristics of the current timer tables are displayed. The command is useful in order to inspect the state of the SDL system, or to see if it is hanging.

Example 622 ----------

After typing the command, information is printed on screen which may look like:

```
Current Timer states:
SDL NOW                    :9147,d
Timer dimensioned to       :6
Currently                  :0 timer(s) active
Timer memory location at   :36bf0014,
```

Where SDL NOW represents the amount of units since system start. The value of 6 shows the maximum number of timer instances in the system. The third line is self explanatory. The last line gives the physical memory allocation of the timer linked lists which may ease debugging.

Tr-Detail

Parameters:

`<level>`

Type: Local

Define Trace detail. The command is applied locally within the SDL Target Tester's host and it works on ASCII traces to the text area only.

Output to either a file or to the MSC Editor will not be affected by this command. Thus if the target is configured correctly no inconsistency can occur and no information stored in a file or in the MSC Editor will be incomplete.

Five levels are defined (0–5). The higher the level, the more SDL events are traced. Different events are assigned the levels depicted in the following table:

Trace subject	L.1	L.2	L.3	L.4	L.5
PRINT STRING	*	*	*	*	*
TASK	-	-	-	-	*
DECISION	-	-	-	-	*
PROCEDURE	-	-	-	-	*
SET	-	-	-	*	*
RESET	-	-	-	*	*
ACTUAL RESET	-	-	-	-	-
STOP	-	-	-	*	*
STATE	-	*	*	*	*
STATIC CREATE	-	-	-	*	*
DYNAMIC CREATE	-	-	-	*	*
CREATE	-	-	-	*	*
SAVE	*	*	*	*	*
TIMER	*	*	*	*	*
INPUT	*	*	*	*	*
OUTPUT	-	-	*	*	*
DISCARD	*	*	*	*	*
IMPLICIT CONSUMPTION	*	*	*	*	*

Tr-Params

Parameters:

`<const>`

Type: Local

The trace of signal parameters can be switched on or off. This command has an influence on the display in the text area only.

<const> can take the values:

0 - to switch trace off or
1 - to switch the trace of signal parameters on

Tr-Process

Parameters:

```
<Processtype Instance> <Bitmask>  or
ENV <Bitmask>  or
* <Bitmask>
```

Type: Remote

This command defines which processes in the system are traced. It is applied in the target (remote command), so that the traffic load on the communications interface can be reduced interactively.

This trace command works for all instances of the given process type. No differentiation is possible between different instances of one type.

Tr-Signal

Parameters:

```
<signalname> <flag>
```

Type: Remote

This command defines the trace for one or all signals in the system. It is applied in the target (remote command), so that the communications interface's traffic load can be reduced interactively.

The signal name is the one from the SDL system. No qualifiers are possible, which means, that signal names have to be unique for the whole SDL system. The flag may be 0 (trace off) or any other value (trace on). If an asterisk is specified for signal name, then flag is applied to all signal types. The command works for SDL output only.

Tr-Off

Parameters:

```
(None)
```

Type: Remote

The command switches the SDL Target Tester's tracer off within the target. The current option settings stored in the target will not be affected. After this command is received within the target, the trace is imme-

diately suspended and can only be resumed with the command Tr-On. The output trace that has been written is interrupted.

> **Note:**
>
> The user has to consider misinterpretation of the trace because of missing traces after resuming the trace via Tr-On.

Tr-On

Parameters:

```
(None)
```

Type: Remote

This command switches the Cmicro Tracer on, if it is compiled within the target. None of the current option settings concerning trace within the target will be affected.

Unit-Name

Parameters:

```
<unitname>
```

Type: Local

The timer unit's name can be assigned with this command, e.g. if the target's system time is 'milliseconds' and the Unit-Scale is dimensioned to 1000, the <unit> should be named as 'sec'.

Unit-Scale

Parameters:

```
(realvalue)
```

Type: Local

The target's system time, which is received with the trace data, is multiplied by this constant factor to get a "readable" system time. See Unit-Name too.

Graphical User Interface

This section describes the appearance and functionality of the graphical user interface of the SDL Target Tester (sdtmtui). Some user interface descriptions common to all tools can be found in chapter 1, *User Interface and Basic Operations*. These general descriptions are not repeated in this chapter.

Starting the SDL Target Tester UI

The SDL Target Tester UI is automatically started from the Cmicro Postmaster. When the UI is started the definition file for the button groups (see "The Button Area" on page 3658) is read in. The default name of this file is `sdtmt.btn`.

The Main Window

The main window provides a text area (which displays output from the monitor system), an input line (used for entering and displaying textual command line input to the monitor system) and a button area (with buttons for execution of monitor commands).

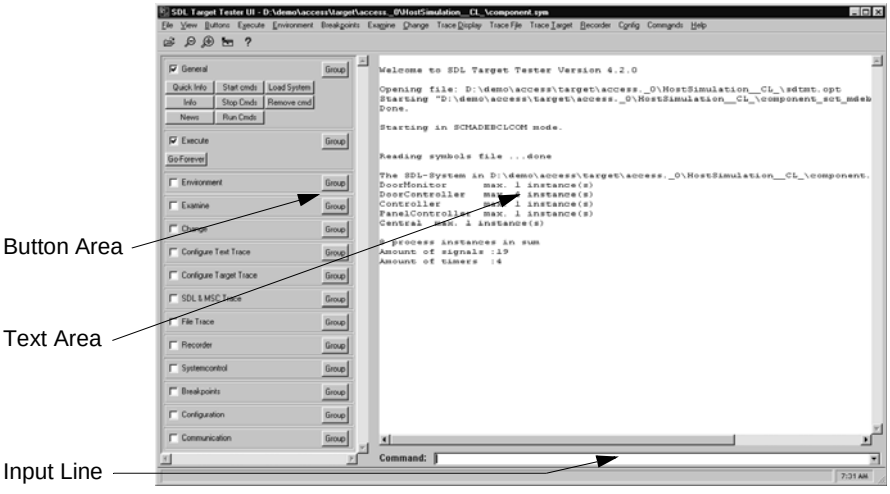


Figure 606: The Main window

The Text Area

The *text area* displays all text output from the monitor system, including user prompts, error messages and command results.

Commands cannot be entered in this area, but a command given on the input line or through the use of the command buttons is echoed after the displayed prompt:

```
Command>
```

The Input Line

The *input line* is used for entering and editing monitor commands from the keyboard. For information on available monitor commands, see "Syntax of SDL Target Tester Commands" on page 3629.

The last commands entered on the input line are saved in a history list. This list can be traversed by using the `<Up>` and `<Down>` keys on the input line.

When `<Return>` is pressed anywhere on the input line, the complete string is saved in the history list and is moved to the text area. The command is then executed.

Parameter Dialogs

If a command entered on the input line requires additional user input (i.e. parameter values), the information will automatically be prompted for in a dialog:

- Parameter values of enumeration type are presented in lists, from which the value can be selected (see Figure 607).

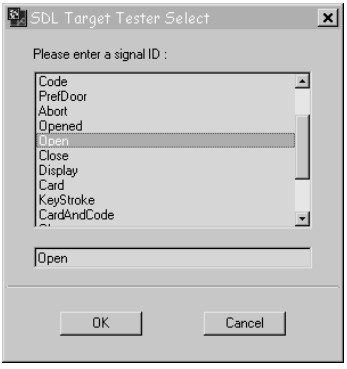


Figure 607: A typical selection dialog

The value can also be entered or edited on the text input line below the list in the dialog. For those commands that take an optional variable component, the component must be entered on the text input line after the selected variable name. It will not be asked for in a dialog.

- Other parameter values are asked for in simple text input dialogs.

Each parameter dialog has an *OK* button for confirming the value and a *Cancel* button for cancelling the command.

The Button Area

The *button area* is used for entering monitor commands by clicking the left mouse button on a command button. Each button corresponds to a specific monitor command. The buttons are divided into groups, roughly corresponding to the different types of commands listed in "SDL Target Tester Commands" on page 3629. Each group is shown as a *module* in the button area. Any number of button modules may reside in the button area. If the modules do not fit in the button area, a vertical scroll bar is added.

The definition of the buttons and button groups are stored in a *button definition file* (see "Button and Menu Definition File" on page 3673). New buttons can be added and existing ones deleted or redefined by using the *Group* menu in a button module.

If a button's definition contains parameters, the parameter values are prompted for in dialog boxes before the command is executed, in the same way as described for commands entered from the input line. See "Parameter Dialogs" on page 3657.

When no more parameter values are requested, the string shown on the input line is saved in the history list and moved to the text area. The command is then executed.

A Button Module

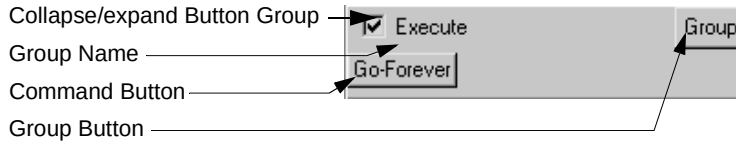


Figure 608: A button module

Each *button module* consists of a title bar and a number of command buttons arranged in rows and columns. The title bar displays:

- A *collapse/expand* toggle button. Click this to collapse (so that only the title bar is visible) and expand the module.
- The *group name* of the button module.
- A *Group* button, providing a menu with commands affecting the buttons in the module.

The *Group* button contains the following menu choices:

Add

This menu choice opens a dialog where you may add one or several new buttons to the button module. You should specify a label of the new button and a command that is to be executed when the button is clicked. Click *OK* or *Apply*, to add the button to the end of the module. For the syntax of a button definition, see "Button and Menu Definitions" on page 3674.

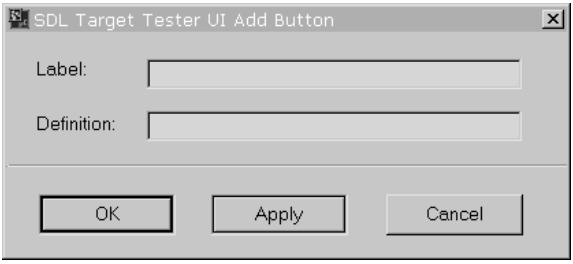


Figure 609: Adding a new button

> **Note:**
> Several buttons are allowed to have the same name within the same module. When you edit the button definition or delete the button, it is always the first occurrence that will be deleted or modified.

Edit

This menu choice opens a dialog where you can select a button to edit, that is, its label and definition. You edit the button in a similar way as described in "Add" on page 3659.

> **Note:**
> If several buttons have **the same button label**, it is always **the first button** found that will be edited, regardless of the selection.

Delete

This menu choice opens a dialog where you can select one or several buttons to be deleted.

> **Note:**
> If several buttons have **the same button label**, it is always **the first button** found that will be deleted, regardless of the selection.

Rename Group

This menu choice opens a dialog where you can edit the name of the current button module.

Delete Group

Select this menu choice deletes to delete the current module from the button area. You will be asked to confirm the operation.

The Default Button Modules

The following tables list the default buttons in the button modules and the corresponding monitor command. See "SDL Target Tester Commands" on page 3629 for more information.

> **Note:**
> The buttons in the button modules are specified in the button definition file. If the default button file is not used, the button modules may differ from those described here. See "Button and Menu Definitions" on page 3674 for more information.

The *Environment* Module

Button	SDL Target Tester command
Send without params	Output-NPAR
Send with params	Output-PAR

The *Execute* Module

Button	SDL Target Tester command
Go-Forever	Go-Forever

The *Examine* Module

Button	SDL Target Tester command
Queue	?Queue
Process	?Process-State
Last Error	?Errors
Timers	?Timer-Table
Process List	?All-Processes

Button	SDL Target Tester command
Active	Active-Timer
Profile	?Process-Profile

The *Change* Module

Button	SDL Target Tester command
Set Timer	Set-Timer
Reset Timer	Reset-Timer
Create	Create
Stop	Stop
Nextstate	Nextstate
Remove Q	Remove-Queue
Remove S	Remove-Signal
Remove all S	Remove-All-Signals
Scale Timers	Scale-Timers

The *Configure Text Trace* Module

Button	SDL Target Tester command
ON	Display-On
OFF	Display-Off
Detail Level	Tr-Detail
Show Params	Tr-Params 1
Hide Params	Tr-Params 0

The *Configure Target Trace* Module

Button	SDL Target Tester command
ON	Tr-On

Button	SDL Target Tester command
OFF	Tr-Off
Signal	Tr-Signal
Process	Tr-Process
NO Symbols	Tr-Process * B'00000000000000000000000000000000'
ALL Symbols	Tr-Process * B'11111111111111111111111111111111'

The *SDL & MSC Trace* Module

Button	SDL Target Tester command
Start MSC	Start-MSC-Log
Stop MSC	Stop-MSC-Log
Start SDL	Start-SDLE-Trace
Stop SDL	Stop-SDLE-Trace

The *File Trace* Module

Button	SDL Target Tester command
Binary Infile	Input-File
Binary Outfile	Output-File
Close binaries	Close-File
ASCII Outfile	Start-Trace-Log
Close ASCII	Stop-Trace-Log
Page File	Page-File
Convert File	Convert-File

The *Recorder* Module

Note:

The SDL Target Tester's Record and Play functions are only available if a Cmicro Recorder license is available.

Button	SDL Target Tester command
Recorder off	Recorder-Off
Record	Recorder-On
Play	Recorder-Play
Delay	Recorder-Delay
Real-time	Recorder-Realtime
Input recfile	Input-File recfile.trc
Output recfile	Output-File recfile.trc
Close recfile	Close-File recfile.trc

The *Systemcontrol* Module

Button	SDL Target Tester command
Single-Step	Single-Step
Next-Step	Next-Step
Exit-Single-Step	Exit-Single-Step
Reinit	Reinitialize
Shutdown	Shutdown
Resume	Resume
Disable Timer	Disable-Timer
Enable Timer	Enable-Timer
Suspend	Suspend

The *Breakpoint* Module

Button	SDL Target Tester command
Break input	BPI
Break nextstate	BPS
Break all	BP
View Breakpoints	?Breaklist
Clear all	BA
Continue	Continue

The *Configuration* Module

Button	SDL Target Tester command
Unit scale	Unit-Scale 1.0
Unit name	Unit-Name “milliseconds”
Target	Get-Configuration

The *Communication* Module

Button	SDL Target Tester command
Start Gateway	Start-Gateway
Coder Config	?Coder
Interface State	Line

The Menu Bar

This section describes the menu bar of the SDL Target Tester main window and all the available menu choices in the default configuration.

The menu bar contains the following fixed menus:

- *File Menu*
- *View Menu*
- *Buttons Menu*

- *Help Menu*
  (See "Help Menu" on page 15 in chapter 1, *User Interface and Basic Operations*.)

The menu bar can also contain additional menus listed in a menu definition file (see "Button and Menu Definition File" on page 3673). The following menus are defined in the delivered version of this file:

- *Execute Menu*
- *Environment Menu*
- *Breakpoint Menu*
- *Examine Menu*
- *Change Menu*
- *Trace Display Menu*
- *Trace File Menu*
- *Trace Target Menu*
- *Recorder Menu*
- *Config Menu*
- *Commands Menu*

File Menu

The *File* menu contains the following menu choices

- *Open*
  (See "Open" on page 9 in chapter 1, *User Interface and Basic Operations*.)
- *Set Directory*
- *Exit*
  (See "Exit" on page 15 in chapter 1, *User Interface and Basic Operations*.)

Set Directory

This menu choice opens a dialog where you can enter the path to your current project directory.

View Menu

The *View* menu contains the following menu choices:

- *Find*
- *Find Next (F3)*

- *Find Selection*
- *Copy from Text Area*
- *Paste into Command Line*
- *Line Numbering*
- *Clear Text Area*

Find

This menu choice opens a dialog where you can search for text.

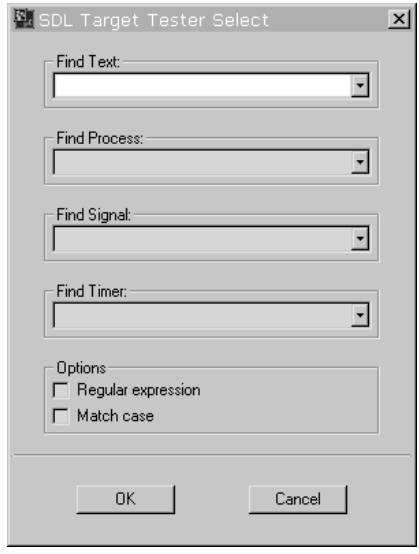


Figure 610: The Find dialog

It is possible to use regular expressions for finding text or to select to find a process, signal or timer belonging to the SDL system.

Find Next

This menu choice will search again for text already found.

Find Selection

This menu choice will search for text currently selected in the SDL Target Tester's text area.

Copy from Text Area

This menu choice will copy the text selected in the SDL Target Tester's text area.

Paste into Command Line

The menu choice will paste the content of the clipboard to the command line.

Line Numbering

This menu choice toggles line numbering in the SDL Target Tester's text area. Line numbering allows a better overview.

Clear Text Area

This menu choice will clear the complete text area of the SDL Target Tester.

Buttons Menu

The *Buttons* menu contains the following menu choices:

- *Load*
- *Append*
- *Save*
- *Save As*
- *Expand Groups*
- *Collapse Groups*
- *Add Group*

For more information on the UI's button definition file, mentioned in the menu commands below, see "Button and Menu Definition File" on page 3673.

Load

This menu choice opens a dialog where you can specify a new button definition file. This will override the current button definitions. All buttons and modules currently in the button area are deleted and replaced by the buttons and modules defined in the new file.

Append

This menu choice opens a dialog where you can specify to append the contents of a new button definition file into the current button definitions. Buttons with new labels are added to the button area, while buttons with already existing labels in the same module will be duplicated (possibly with different definitions).

Save

This menu choice saves the current button and module definitions in the button definition file under its current file name.

Save As

This menu choice opens a dialog where you can save the current button and module definitions in a new button definition file.

Expand Groups

This menu choice expands all modules in the button area.

Collapse Groups

This menu choice collapses all modules in the button area.

Add Group

This menu choice opens a dialog where you can specify the of a new button module to add. The new module will be added after the last module in the button area. It is also possible to add several modules.

Execute Menu

Item	SDL Target Tester command
Start Gateway	Start-Gateway
Go-Forever	Go-Forever
Single-Step	Single-Step
Reinit	Reinitialize
Disable-Timer	Disable-Timer
Next-Step	Next-Step

Item	SDL Target Tester command
Shutdown	Shutdown
Enable Timer	Enable-Timer
Exit-Single-Step	Exit-Single-Step
Resume	Resume
Suspend	Suspend

Environment Menu

Item	SDL Target Tester command
Send with params	Output-PAR
Send without params	Output-NPAR

Breakpoint Menu

Item	SDL Target Tester command
Break Input	BPI
View Breakpoints	?Breaklist
Break Nextstate	BPS
Clear all	BA
Break all	BP
Continue	Continue

Examine Menu

Item	SDL Target Tester command
Queue	?Queue
Timers	?Timer-Table
Active	Active-Timer
Process	?Process-State

Item	SDL Target Tester command
Process List	?All-Processes
Profile	?Process-Profile
Last Error	?Errors

Change Menu

Item	SDL Target Tester command
Set Timer	Set-Timer
Stop	Stop
Remove Signal	Remove-Signal
Reset Timer	Reset-Timer
Nextstate	Nextstate
Remove all S	Remove-All-Signals
Create	Create
Remove Q	Remove-Queue
Scale Timers	Scale-Timers

Trace Display Menu

Item	SDL Target Tester command
Display on	Display-On
Display off	Display-Off
Trace Detail	Tr-Detail
Show Params	Tr-Params 1
Hide Params	Tr-Params 0
Start MSC	Start-MSC-Log
Stop MSC	Stop-MSC-Log

Trace File Menu

Item	SDL Target Tester command
Binary Infile	Input-File
ASCII Outfile	Start-Trace-Log trace.asc
Page File	Page-File
Binary Outfile	Output-File
Close ASCII	Stop-Trace-Log
Convert File	Convert-File
Close Binaries	Close-File

Trace Target Menu

Item	SDL Target Tester command
Trace on	Tr-On
Trace off	Tr-Off
Trace Process	Tr-Process
Trace Signal	Tr-Signal

Recorder Menu

Item	SDL Target Tester command
Recorder off	Recorder-Off
Delay	Recorder-Delay
Output recfile	Output-File recfile.trc
Record	Recorder-On
Realtime	Recorder-Realtime
Close Recfile	Close-File recfile.trc
Play	Recorder-Play
Input recfile	Input-File recfile.trc

Config Menu

Item	SDL Target Tester command
Unit scale	Unit-Scale
Unit name	Unit-Name
Target	Get-Configuration
Coder-Config	?Coder
Interface state	Line

Commands Menu

Item	SDL Target Tester command
Quick Info	Help
Info	Help *
Remove cmd	Remove-Command

Button and Menu Definition File

In the SDL Target Tester UI, the button definition information is stored on file (per default this file is named `sdtmt.btn`), i.e. definitions of button groups and button commands in the main window's button area. Furthermore it is possible to define menu entries.

At start-up of the UI, the file is determined in the following way. If the file name does not contain a directory path, the files are searched for in the following directories:

- The current directory
- The SDL Suite installation directory

Once the file has been found, it is read and the contents of the corresponding window is set up. If the file cannot be found, the corresponding window area becomes empty and a file selection dialog is opened to enter the required file name.

The text file can contain comment lines starting with the `'#'` character. Empty lines are discarded.

Note:

When a file is read, no checks are made upon the relevance or correctness of the definitions contained in the file.

Button and Menu Definitions

The *button and menu definitions* are stored in a definition file with the default extension `.btn`. The definitions are divided into groups where each group defines a button module in the main window's button area or a menu in the menu bar.

Syntax

In the file, a button group has the following syntax:

```
:[DISPLAY_AS] <entry name>
<item label>
<definition>
<button label>
<definition>
  . . .
```

The [DISPLAY_AS] prefixes the button group's or menu's name <entry name> and can take three different keywords:

- `COLLAPSED`
  A button group is created, initially collapsed.
- No keyword
  A button group is created, initially expanded.
- `MENU`
  A new menu is created.

The <item label> is either the label of the button in a button module or the label of an menu item.

The <definition> is the SDL Target Tester command that will be executed when the button is pressed/the menu item is selected. The syntax of a button definition is the same as when entering a command textually to the monitor.

Missing parameters at the end of the command will open dialogs for those parameters.

After all there is no limit in the amount of buttons in a button group or the amount of items in a menu entry except the visibility and usability of the buttons/items.

Comparing with the Simulator there is a simple rule of ordering the buttons shown in Figure 611. A new button row is added only if the previous row has got 3 buttons.

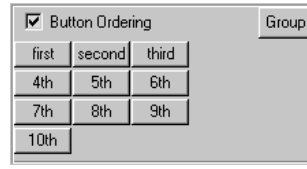


Figure 611: Ordering of the buttons in a button group

The Target Library

General

The target library is a mandatory part of the SDL Target Tester. The SDL Target Tester is of no use without the functions and definitions contained within.

In order to use the options of SDL Target Tester's host site, the target library must be configured accordingly.

A lot of defines make it possible to reduce the SDL Target Tester's functions within the target, so that memory requirements can be reduced when necessary.

In the following sections are explanations of the file structure and the application program interface (API) within the target.

File Structure

The following picture explains the file structure for the SDL Target Testers target library.

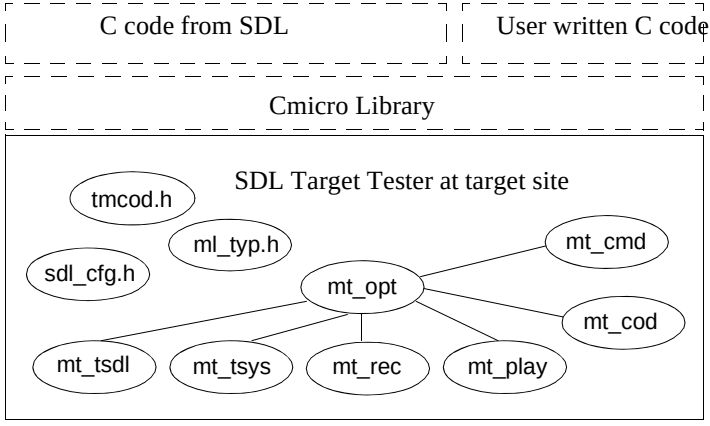


Figure 612: Files of the Cmicro Library

Description of Files

sdl_cfg.h

This file is generated automatically by the Cmicro SDL to C Compiler into the directory which is currently active. It contains compilation flags used for the automatic scaling of the Cmicro Library and the generated C code. The file must not be edited by the user.

> **Caution!**
>
> This file always carries the same name for each SDL system generated and is stored into the currently active directory. A conflict will occur if the user tries to generate several systems into the same directory.
>
> Therefore be sure to use different working directories for each SDL system. Otherwise, unpredictable results during run-time of the generated code will occur, as some automatic scalable features may not re-compiled.

ml_typ.h

This file is the central header file in the Cmicro Package. It contains

- more #includes
- defines which are of global interest and Cmicro Library internal defines
- typedefs which are of global interest and Cmicro Library internal typedefs
- extern-declarations which are of global interest and Cmicro Library internal extern declarations

tmcod.h

`tmcod.h` is an automatically generated file of the Target Message Coder Generator (TMCOD). All the SDL Target Tester command tags and the corresponding parameter structures are defined here. It is strongly recommended not to modify this file.

mt_cmd.c

This file is part of the command interface on the target site. The commands received via the communications link will be performed by this file.

mt_cod.c

This is the module which exports the functions to encode / decode any message from the Cmicro Protocol.

mt_opt.c

This module exports the functions which allow the user to define options regarding the tracer, recorder and player components.

Options may be specified to reduce the traffic load of the system, This is important when a slow communications interface is involved and the user wants to test in real-time.

mt_play.c

This file is part of the Cmicro Recorder and it contains all the functions for the play mode.

mt_rec.c

This file is part of the Cmicro Recorder and it contains all the functions for the record mode.

mt_tsdl.c

This module exports functions to trace SDL actions. These functions are mainly for the use of the Cmicro Kernel and Tester.

Functions to trace either into RAM, into a file or via a V.24 interface are delivered.

The module is common for all these trace methods. The functions exported are called directly by the Cmicro Kernel if the SDL Target Tester or part of the SDL Target Tester is involved when executing the SDL system.

mt_tsys.c

This module contains the functions which handle the trace of system events such as scheduler events or the trace of system errors.

mt_*.h

These header files contain the external definitions of the corresponding `.c` files.

The API on Target

General

The C application program interface (API) can be used both by SDL processes and / or by any other C function on the target.

The simplest method, however, is to set some options before initializing the SDL system and hold these options during the whole run-time. A more comfortable method is to set some options before initializing the SDL system and then change the options in an SDL process, when a specific situation is detected. By using this method it is possible to reduce the amount of information and to make it easier to find unwanted behavior or an erroneous situation in a trace. This can be achieved, for example, by writing the necessary C function call(s) into an SDL task symbol using the #CODE directive.

The following pages introduce the C function interface, which is used in the target to set the trace options.

> **Caution!**
>
> It is important to take care with function parameters. Any function return value should also be checked.

Initialization

```
void xmk_InitOptions ( void )
```

Initialization of the SDL Target Tester is absolutely necessary. It must be done before any other initialization, directly before the C function call `xmk_InitSDL`. The user should not remove this function call in the template `main()` function if one wants to use the SDL Target Tester.

Nothing happens before the C function `xmk_OptionsSymbol` is called, and `xmk_TracerActive` is set to `XMK_TRUE`.

Switch the Cmicro Tracer on / off

In order to get an execution trace it is first necessary to switch it on by assigning

```
xmk_TracerActive = XMK_TRUE
```

This should normally be done after the C function `xmk_InitOptions`.

The settings of functions whose names begin with `xmk_Options*` remains unaffected when changing the above flag.

The flag is **not** set by default.

Selecting SDL Symbols That are to Be Traced

```
xmk_OPT_INT xmk_OptionsSymbol ( unsigned char process-type-id,
                                long bitmask )
```

This function must be called in order to begin the trace. By default, no symbol is selected.

Select the trace for process-type-id and then set or un-set different symbols with the bit mask. If anything goes wrong the function will return `XMK_ERROR`, else `XMK_OKAY`.

It is possible to specify bit mask values for a specific SDL process, for all SDL processes, for the environment and for the kernel part of the Cmicro Library. The last possibility is necessary to allow the optional trace for system events like error trace and trace of scheduling events.

Allowed values for process-type-id:

0	First process-type-id, XPTID_ from `sdl_cfg.h`
1	Second.....
N-1	Last... (N is the maximum number of process types)
xNULLTYPE	specify trace for all processes
any other value	The environment
MICROKERNEL	specify trace for Cmicro Kernel
ENV	Indicates the SDL environment

Allowed values for bit mask:

SDL symbols	System events
TSDL_STATE	TSYS_SCHEDULE
TSDL_INPUT	TSYS_ERROR
TSDL_SAVE	TSYS_SHOWPRIO
TSDL_OUTPUT	
TSDL_CREATE	
TSDL_STOP	
TSDL_STATIC_CREATE	
TSDL_DYNAMIC_CREATE	
TSDL_DECISION	
TSDL_TASK	
TSDL_PROCEDURE	
TSDL_RESULT	
TSDL_TIMER	
TSDL_SET	
TSDL_RESET	
TSDL_ACTUAL_RESET	
TSDL_SIGNALPARAMS	
TSDL_DISCARD	
TSDL_IMPLICIT_CONSUMPTION	

Selecting SDL Signals to Be Traced

```
xmk_OPT_INT  xmk_OptionsSignal ( xmk_T_SIGNAL signal id,
                                 unsigned char bitmask )
```

This function is optionally called because the trace of all SDL signals is switched on per default.

Select the trace for signal id. Then set or un-set different symbols with bit mask. If anything went wrong, the function will return `XMK_ERROR`, else `XMK_OKAY`.

It is possible to specify either “off” for all signals, “on” for all signals, “off” for a specific signal or “on” for a specific signal.

Allowed values for signal-id:

0	First signal-id, #define from generated C code and `<systemname>.ifc`
1	Second.....
N-1	Last.....
XMK_ALLOW_NO	Specify trace for all processes
XMK_ALLOW_ALL	Specify trace for Cmicro Kernel

Allowed values for bitmask:

- Any value in the range of `0x00` to `0xff`.
- A value != `0x00` switches the trace for the signal on.
- A value == `0x00` causes the trace for the given signal(s) to be switched off.

Example to Set Trace Options

A simple method to set the trace options is to set them directly within the C function `main`, before the Cmicro Kernel is initialized. The user will find this example as a template in the file `bare_ex.c` delivered with the Cmicro Package, below the directory `<sdtdir>/cmicro/template`.

Example 623 ----------

```
#include "sdl_cfg.h"
#include "ml_typ.h"
    ... main ( ... )    {
  extern int xmk_TracerActive ;
  long       bitmask = 0      ;
  int        result  = XMK_OKAY ;

/* Initialize SDL Target Tester */
xmk_InitOptions ( );

/* Set all options ON */
result = xmk_OptionsSymbol (xNULLPID, 0xffffffff);
if (result != XMK_OKAY) ErrorInUsage ();

/* Set some options ON */
bitmask = TSDL_INPUT  ;
bitmask = TSDL_STATE  ;
bitmask = TSDL_OUTPUT ;

                /*          +---- Proces-Type - Id (see automatically */
                /*          |     generated file *.ifc)               */
                /*          V                                         */
 result = xmk_OptionsSymbol (0, bitmask );
 if (result != XMK_OKAY) ErrorInUsage ();
 result = xmk_OptionsSymbol (1, 0x00000000 );
 if (result != XMK_OKAY) ErrorInUsage ();

/* S w i t c h   o n   t h e   t r a c e */
xmk_TracerActive                = XMK_TRUE ;
/* Template to set options to filter signals */
                   /*                                              */
                   /* +---- Allow all Signals to be traced       */
                   /* V                                           */
 xmk_OptionsSignal (XMK_ALLOW_NO, 0);
                /* +---- Signal id is to be set correctly !    */
                /* |     according to the automatically       */
                /* |     generated *.ifc - file               */
                /* V                                          */
 xmk_OptionsSignal (1, 0xff);
 xmk_OptionsSignal (2, 0xff);
 xmk_OptionsSignal (3, 0xff);
/* I n i t i a l i z e   C m i c r o   K e r n e l */
xmk_InitSDL ()

/* R u n   C m i c r o   K e r n e l */
if (xmk_RunSDL (0xff) != XMK_STOP);
exit (0);
}
```

Getting a Trace in SDL Tasks with C Code

The Cmicro SDL to C Compiler generates usable traces for SDL tasks only if it is a task containing at least one SDL assignment. Whenever C code is specified within a task symbol (by using the #CODE directive), the Cmicro SDL to C Compiler is not able to generate good trace information.

There are two C functions which enable a work-around:

- `xmk_PrintString (const char *)` or
- `xmk_TSDL_Task (const char *)`

in order to get a trace from C code. The trace output of a call like

```
xmk_PrintString ('Hello SDL world')
```

looks like:

```
USER:   Hello SDL world
```

The strings have to be unique, in order to distinguish a trace of one process from another. Another difference between the above function is that the C function `xmk_PrintString` does not check if a trace is wanted or not. The trace will be done unconditionally via the SDL Target Tester's communications link.

Of course, the function may also be applied in non SDL program parts.

Using Record or Play Mode of the Cmicro Recorder

```
xmk_OPT_INT xmk_SetRecorderMode ( int mode )
```

This function must be called by the user in order to:

- switch the record mode of the Cmicro Recorder on, or
- switch the play mode of the Cmicro Recorder on, or
- switch any function of the Cmicro Recorder off.

Allowed values for mode:

- `XMK_RECORDER_RECORD`
- `XMK_RECORDER_PLAY`
- `XMK_RECORDER_OFF`

Environment Functions

xInEnv ()

```
void  xInEnv ( void )
```

This function is called each time the Cmicro Kernel's main loop is entered. A more detailed description of the C function `xInEnv()` can be found in "xInEnv()" on page 3533 in chapter 66, *The Cmicro Library*.

xOutEnv ()

```
int  xOutEnv ( xmk_T_SIGNAL sig,
               xmk_T_PRIO prio,
               xmk_T_MESS_LENGTH data_len,
               void *p_data,
               xPID Receiver
```

This function is called each time the SDL system wants to output a signal to the environment. If the environment should be another system (e.g., an MS DOS PC), the signal can be sent directly by calling the C function `xmk_Cod_Encode` (which is described in the following section). A more detailed description of the C function `xOutEnv` can be found in "xOutEnv()" on page 3536 in chapter 66, *The Cmicro Library*.

xmk_Cod_Decode ()

```
int  xmk_Cod_Decode ( char *p_RAWBuffer,      In
                      int *p_MessageClass,    Out
                      int *p_MessageTag,      Out
                      int *p_MessageLength,   Out
                      char *p_dest,           In/Out
                      int dest_len )          In
```

This function is free to be used on the target site. It decodes a given data link frame in the Cmicro Protocol format into the different pieces of data. `MessageClass` may be in the range from `F3h` to `FFh`. The value `Fh` in the higher nibble is used to synchronize the data stream if synchronization fails and `0h` to `2h` in the lower nibble is used by the SDL Target Tester itself. `MessageTag` may be of any value. `p_struct` is a pointer to a C structure which is to be transferred via the communications link. `Struct_length` is the result of the sizeof operator being applied to this structure.

The function is the counterpart to the C function `xmk_Cod_Encode`.

xmk_Cod_Encode ()

```
int  xmk_Cod_Encode ( int MessageClass,
                      int MessageTag,
                      char *p_data,
                      int length )
```

This function is free to be used on the target site: It encodes the given information into a data link frame of the Cmicro Protocol format and sends it via the communications interface. `MessageClass` may be in the range from `F3h` to `FFh`, the value `Fh` in the higher nibble is used to synchronize the data stream if synchronization fails and `0h` to `2h` in the low-

er nibble is used by the SDL Target Tester itself. `MessageTag` may be of any value. `p_struct` is a pointer to a C structure which is to be transferred via the communications link. `Struct_length` is the result of the sizeof operator being applied to this structure.

The function is the counterpart to the C function `xmk_Cod_Decode`.

> **Note:**
> The maximum amount of characters which can be transferred is restricted to 255 per default in this Cmicro Package version.

Compiling and Linking the Target Library

Some defines are to be set in order to compile and link the parts of the SDL Target Tester correctly. These defines can be set either on the command line when invoking make or in the header file `ml_mcf.h`. Be sure that for each configuration possibility all the defines are set appropriately.

The best way to select the compiler options for the SDL Target Tester is to use the Targeting Expert. Please view the section "Configure the SDL Target Tester (Cmicro only)" on page 2947 in chapter 59, *The Targeting Expert*.

Here is a first idea of what has to be to set in the configuration file `ml_mcf.h`:

```
#define XMK_ADD_MICRO_TESTER
#define XMK_ADD_MICRO_TRACER
#define XMK_USE_V24
#define XMK_USE_COMMLINK
```

Scaling of SDL Target Tester Functionality

In order to reduce the amount of memory - e.g. if memory gets scarce within a progressing project - it is possible to reduce the SDL Target Tester's functionality on target site.

Removing the whole SDL Target Tester

It is quite simple to remove any functionality used by the SDL Target Tester by specifying:

```
#undef XMK_ADD_MICRO_TESTER
```

Removing the Cmicro Tracer

This is possible by specifying

```
#define  XMK_ADD_MICRO_TESTER
#undef   XMK_ADD_MICRO_TRACER
```

Removing the Cmicro Recorder (Inclusive Play Mode)

This is possible by specifying

```
#define  XMK_ADD_MICRO_TESTER
#undef   XMK_ADD_MICRO_RECORDER
```

Removing the Command and Debug Interface

This is possible by specifying

```
#define  XMK_ADD_MICRO_TESTER
#undef   XMK_ADD_MICRO_COMMAND
```

Partial functionality may be left out by un-defining one of the defines described under "Message Transfer and Presentation of Messages" on page 3689.

Configuration of Buffers

Users must configure buffers used by the SDL Target Tester.

The reason behind this is that the SDL Target Tester uses an efficient way to handle those buffers. This is of great importance especially for micro controllers.

Additionally, these buffers must be consistently dimensioned. For instance, the message buffer of the communications link must be able to handle the largest parameter list of a signal from SDL. The same situation arises in the trace of SDL tasks and procedure names.

Caution!

If any buffer is dimensioned too small a run-time error may result.

The following defines are used to dimension buffers.

XMK_MAX_PRINT_STRING

This define restricts the amount of characters transferred via the communications link when using the C function `xmk_PrintString`. This function is used only by the user.

XMK_MAX_LEN_SYMBOL_NAME

This define is used to restrict the amount of characters transferred via the communications link for the SDL tasks and SDL procedure names.

The transmit and receive buffers of the default communications link software must be configured in the data link (`dl`) module using the following defines:

Transmitter buffer:

```
#define XMK_MAX_SEND_ONE_ENTRY          1000
#define XMK_MAX_SEND_ENTRIES            20
```

Receiver buffer:

```
#define XMK_MAX_RECEIVE_ONE_ENTRY       1000
#define XMK_MAX_RECEIVE_ENTRIES         20
```

where `*_ONE_ENTRY` is the dimensioning of one message, and `*_ENTRIES` is the maximum amount of entries in the message FIFO of the data link module. Note, that `*_ONE_ENTRY` must be greater than the greatest of the `XMK_MAX_*` defines above, plus a reserve of 6 bytes.

Connection of Host and Target

General

The following sections will detail the technical requirements and the technical implementation of the connection between host and target. The following items will be outlined:

- "Structure of Communications Link Software" on page 3689
- "Default Implementation of Communications Link Software" on page 3689 as delivered.
- How to implement user host executables, which are able to communicate with the SDL Target Tester's target library. This is described in the subsection "Open Interface" on page 3726.

Structure of Communications Link Software

This is the structure for the communications link software, which is - in principle - applicable for host, as well as for target site:

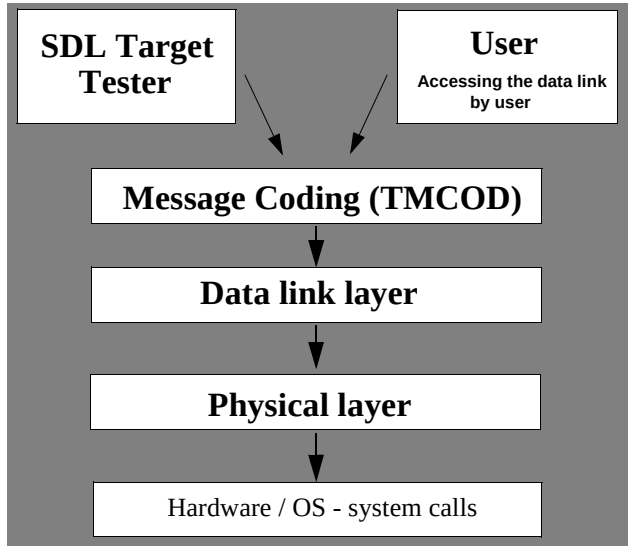


Figure 613: Structure of the communications link software

The following subsections describe the current implementation and how to implement a new communications link software.

Default Implementation of Communications Link Software

Message Transfer and Presentation of Messages

State Machine and Handshake

For all the commands and messages, there is a small state machine to handle the correct transfer of information. All messages, which have the suffix `_REQ` (request) in their name (message tag) are to be confirmed from the target with one of the `_CNF` (confirmation) messages. (See the file `tmcod.h`, where all message tags are defined).

For messages which take parameters, there is always (no exception) a special _CNF message which carries the response parameters back to the requestor. The requestor is usually the host.

For the messages, which do not carry parameters, a positive confirmation is done via the message CMD_OKAY_CNF.

If a message cannot be recognized or processed within the target, a negative confirmation CMD_ERROR_CNF is used in all cases.

Each _REQ message must be confirmed before the next one can be sent.

Messages with an _IND (indication) suffix are mostly sent from target to host in order to indicate a specific situation. For these messages, the sender is always the originator.

Message Formats

The following subsections give a complete list of messages which can be sent to and received from the target.

Each typedef shown below is packed into a data link frame in its target representation.

That means a memcpy (destbuffer,struct,sizeof(struct)) is performed to copy a structure to/from the data link frame.

On host site, it is therefore necessary to perform message encoding and message decoding.

Formats Concerning the Cmicro Tracer

All messages belonging to the Cmicro Tracer are part of the message class XMK_MICRO_TRACER (F1h).

- Message tag: CMD_TTASK

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Task
Sent when          :Execution of that SDL symbol
Direction          :T->H
Typedefinition     :xmk_T_CMD_TTASK
Conditional compile:XMK_ADD_TTASK
```

  Message description: The message is sent when a task symbol is entered in generated C Code and when the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_TSET

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Set
Sent when          :Execution of that SDL symbol
Direction          :T->H
Typedefinition     :xmk_T_CMD_TSET
Conditional compile:XMK_ADD_TSET
```

Message description: The message is sent when a timer set is performed in generated C Code and when the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_TRESET

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Reset
Sent when          :Execution of that SDL symbol
Direction          :T->H
Typedefinition     :xmk_T_CMD_TRESET
Conditional compile:XMK_ADD_TRESET
```

Message description: The message is sent when a timer reset is performed in generated C Code and when the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_TSTOP

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Stop
Sent when          :Execution of that SDL symbol
Direction          :T->H
Typedefinition     :xmk_T_CMD_TSTOP
Conditional compile:XMK_ADD_TSTOP
```

Message description: The message is sent when an SDL process stop is performed and when the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_TSTATE

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL State
Sent when          :Execution of that SDL symbol
Direction          :T->H
Typedefinition     :xmk_T_CMD_TSTATE
Conditional compile:XMK_ADD_TSTATE
```

Message description: The message is sent when an SDL nextstate is performed and when the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_TSTATIC_CREATE

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Static create
Sent when          :Execution of that SDL symbol
Direction          :T->H
Typedefinition     :xmk_T_CMD_TSTATIC_CREATE
Conditional compile:XMK_ADD_TSTATICCREATE
```

Message description: The message is sent for each statically created SDL process for which the start transition is executed and for which the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_TCREATE

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Create
Sent when          :Execution of that SDL symbol
Direction          :T->H
Typedefinition     :xmk_T_CMD_TCREATE
Conditional compile:XMK_ADD_TCREATE
```

Message description: The message is sent when the SDL action create is performed and when the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_TDYNAMIC_CREATE

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Dynamic create
Sent when          :Execution of that SDL symbol
Direction          :T->H
Typedefinition     :xmk_T_CMD_TDYNAMIC_CREATE
Conditional compile:XMK_ADD_TDYNAMICCREATE
```

Message description: The message is sent for each dynamically created SDL process, when its start transition is executed, and when the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_TDECISION

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Decision
Sent when          :Execution of that SDL symbol
Direction          :T->H
Typedefinition     :xmk_T_CMD_TDECISION
Conditional compile:XMK_ADD_TDecisionValue
```

Message description: The message is sent when an SDL decision is executed and the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_TSAVE

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Save
Sent when          :Execution of that SDL symbol
Direction          :T->H
Typedefinition     :xmk_T_CMD_TSAVE
Conditional compile:XMK_ADD_TSAVE
```

Message description: The message is sent when a signal save is performed and when the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_TTIMER

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Timer
Sent when          :Execution of that SDL symbol
Direction          :T->H
Typedefinition     :xmk_T_CMD_TTIMER
Conditional compile:XMK_ADD_TTIMER
```

  Message description: The message is sent when an SDL timer has expired and is input into the owner process and the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_TINPUT

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Input
Sent when          :Execution of that SDL symbol
Direction          :T->H
Typedefinition     :xmk_T_CMD_TINPUT
Conditional compile:XMK_ADD_TINPUT
```

  Message description: The message is sent when an SDL input is performed and when the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_TPROCEDURE

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Procedure
Sent when          :Execution of that SDL symbol
Direction          :T->H
Typedefinition     :xmk_T_CMD_TPROCEDURE
Conditional compile:XMK_ADD_TNAME
```

  Message description: The message is sent when an SDL procedure is entered and when the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_TOUTPUT

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Output
Sent when          :Execution of that SDL symbol
Direction          :T->H
Typedefinition     :xmk_T_CMD_TOUTPUT
Conditional compile:XMK_ADD_TOUTPUT
```

  Message description: The message is sent when an SDL output is performed and the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_TDISCARD

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Discard
Sent when          :Execution of that SDL symbol
Direction          :T->H
```

```
Typdefinition      :xmk_T_CMD_TDISCARD
Conditional compile:XMK_ADD_TDISCARD
```

Message description: The message is sent when an SDL discard has been detected and when the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_PRINT_STRING

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :User command Print string
Sent when          :Execution of fct.xmk_PrintString()
Direction          :T->H
Typedefinition     :xmk_T_CMD_PRINT_STRING
Conditional compile:XMK_ADD_CPRINT_STRING
```

Message description: The message is sent unconditionally, when the C function `xmk_PrintString` is called by the user.

- Message tag: CMD_TIMPLICIT_CONSUMPTION

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Implicit consumption
Sent when          :Implicit consumption occurs
Direction          :T->H
Typdefinition      :xmk_T_CMD_TIMPLICIT_CONSUMTION
Conditional compile:XMK_ADD_TIMPLICIT_CONSUMPTION
```

Message description: The message is sent when an SDL implicit consumption is performed and when the trace is set for this symbol. Furthermore, the trace must be active.

- Message tag: CMD_TACTUAL_RESET

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of SDL Actual reset
Sent when          :SDL timer is actually reset
Direction          :T->H
Typedefinition     :xmk_T_CMD_TACTUAL_RESET
Conditional compile:XMK_ADD_TACTUAL_RESET
```

Message description: The message is sent when an SDL timer is actually reset and the trace is set for this symbol. Furthermore, the trace must be active. An actual reset applies only if a timer was previously set and has not expired.

- Message tag: CMD_TSYS_ERROR

```
Messageclass       :XMK_MICRO_TRACER
Meaning            :Trace of system error
Sent when          :Any kind of systemerror was detected
Direction          :T->H
Typedefinition     :xmk_T_CMD_TSYS_ERROR
Conditional compile:XMK_ADD_TERROR
```

Message description: The message is sent when a system error of any type has been detected and the trace is set for this event. Furthermore, the trace must be active.

- Message tag: CMD_TSYS_SCHEDULE

```
Messageclass        :XMK_MICRO_TRACER
Meaning             :Trace of system schedule event
Sent when           :When scheduling is performed
Direction           :T->H
Typedefinition      :xmk_T_CMD_TSYS_SCHEDULE
Conditional compile:XMK_ADD_TSCHEDULE
```

Message description: The message is sent when the Cmicro Kernel begins scheduling and the trace is set for this event. Furthermore, the trace must be active.

- Message tag: CMD_TSYS_SHOWPRIO

```
Messageclass        :XMK_MICRO_TRACER
Meaning               :Trace of system event : change priority
level
Sent when           :When changing to another prioritylevel
Direction           :T->H
Typedefinition      :xmk_T_CMD_TSYS_SHOWPRIO
Conditional compile:XMK_ADD_TSHOWPRIO
```

Message description: The message is sent when the Cmicro Kernel schedules from one priority level to another and the trace is set for this event. Furthermore, the trace must be active.

- Message tag: CMD_TBETWEEN_SYMBOL

```
Messageclass        :XMK_MICRO_TRACER
Meaning             :Another symbol than input is executed
Sent when           :Another symbol than input is executed
Direction           :T->H
Typedefinition      :xmk_T_CMD_TBETWEEN_SYMBOL
Conditional compile:XMK_ADD_SDLE_TRACE
```

Message description: With this command the process ID of the current executed process is sent and a system wide unique identifier. The SDL Target Tester can force the SDL Editor to highlight the current SDL symbol.

- Message tag: CMD_TAT_FIRST_SYMBOL

```
Messageclass        :XMK_MICRO_TRACER
Meaning             :An Input symbol has bee executed
Sent when           :An Input symbol is executed
Direction           :T->H
Typedefinition      :xmk_T_CMD_TAT_FIRST_SYMBOL
Conditional compile:XMK_ADD_SDLE_TRACE
```

Message description: Similar like CMD_TBETWEEN_SYMBOLS the currently executed SDL input symbol will be highlighted in the SDL Editor.

Formats Concerning the Cmicro Recorder

All messages belonging to the Cmicro Recorder are part of the message class XMK_MICRO_RECORDER (F2h).

- Message tag: CMD_RECORD_COUNT_IND

```
Messageclass        :XMK_MICRO_RECORDER
Meaning             :Show recorded event counter to host
Sent when           :When ENV->SDL stimuli detected
Direction           :T->H
Typedefinition      :xmk_CMD_RECORD_COUNT_IND
Conditional compile:XMK_ADD_MICRO_RECORDER
```

Message description: The message is sent when the Cmicro Recorder is active and is in record mode. It is sent together with, but always before CMD_RECORD_OUTPUT_ENV2SDL.

- Message tag: CMD_RECORD_OUTPUT_ENV2SDL

```
Messageclass        :XMK_MICRO_RECORDER
Meaning             :Indicate stimuli ENV->SDL to host
Sent when           :When ENV->SDL stimuli detected
Direction           :T->H
Typedefinition      :xmk_T_CMD_RECORD_OUTPUT_ENV2SDL
Conditional compile:XMK_ADD_MICRO_RECORDER
```

Message description: The message is sent when the Cmicro Recorder is active and is in record mode. It is sent after CMD_RECORD_COUNT_IND.

- Message tag: CMD_RECORD_OUTPUT_SDL2ENV

```
Messageclass        :XMK_MICRO_RECORDER
Meaning             :unused
Sent when           :-
Direction           :T->H
Typedefinition      :-
Conditional compile:XMK_ADD_MICRO_RECORDER
```

Message description: This message is currently unused and is never sent.

- Message tag: CMD_PLAY_SECTION_REQ

```
Messageclass        :XMK_MICRO_RECORDER
Meaning             :Request next sections from host
Sent when           :When current cnt reaches debit cnt
Direction           :H->T
Typedefinition      :-
Conditional compile:XMK_ADD_MICRO_RECORDER
```

Message description: The message is sent when the Cmicro Recorder is active and is in play mode. It is sent if the target has to put in the next stimuli (an environment signal).

- Message tag: CMD_PLAY_COUNT

```
Messageclass       :XMK_MICRO_RECORDER
Meaning            :Transfer Debitcounter to target
Sent when          :When target requests next sections
Direction          :H->T
Typedefinition     :xmk_T_CMD_PLAY_COUNT
Conditional compile:XMK_ADD_MICRO_RECORDER
```

Message description: The message is sent when the Cmicro Recorder is active and is in play mode. It is sent if the target requests the next stimuli (an environment signal). It is also sent in the initialization phase of the play mode. Outside the initialization phase, it is always sent together with, but after `CMD_PLAY_OUTPUT_ENV2SDL`.

- Message tag: CMD_PLAY_OUTPUT_ENV2SDL

```
Messageclass       :XMK_MICRO_RECORDER
Meaning            :Transfer stimuli ENV->SDL to target
Sent when          :When target requests next sections
Direction          :H->T
Typedefinition     :xmk_T_CMD_PLAY_OUTPUT_ENV2SDL
Conditional compile:XMK_ADD_MICRO_RECORDER
```

Message description: The message is sent when the Cmicro Recorder is active and is in play mode. It is sent together with `CMD_PLAY_COUNT` but always before `CMD_PLAY_COUNT` if the target request the next stimuli (an environment signal).

- Message tag: CMD_PLAY_OUTPUT_SDL2ENV

```
Messageclass       :XMK_MICRO_RECORDER
Meaning            :unused
Sent when          :-
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_MICRO_RECORDER
```

Message description: This message is currently unused and is never sent.

- Message tag: CMD_RECORDER_OFF

```
Messageclass       :XMK_MICRO_RECORDER
Meaning            :Switch off recorder
Sent when          :When at end of replay session detected
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_MICRO_RECORDER
```

Message description: This message is sent if the host detected an end of file after the target has requested the next stimuli (an environ-

ment signal). It switches the Cmicro Recorder within the target off and execution returns from simulation to real-time.

Formats Concerning the Command and Debug Interface

All messages belonging to the Cmicro Commands are part of the message class `XMK_MICRO_COMMAND (F0h)`.

- Message tag: CMD_SUSPEND_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Suspend Cmicro Kernel
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CSUSPEND
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_RESUME_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Resume  Cmicro Kernel
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CRESUME
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_SYSTEM_REINIT_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Reinitialize SDL system
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CREINIT
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_SYSTEM_SHUTDOWN_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Shutdown SDL system
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CSHUTDOWN
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_SINGLE_STEP_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Go into single step
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CSINGLE_STEP
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_DISABLE_TIMER_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Disable processing of timers
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CDISABLE_TIMER
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_ENABLE_TIMER_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Enable processing of timers
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CENABLE_TIMER
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_NEXT_STEP_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Execute next step
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CNEXT_STEP
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_SYSTEM_CONTINUE_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Continue system, if halted
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CCONTINUE
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_SET_BREAKPOINT_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Set breakpoint
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T__CMD_SET_BREAKPOINT_REQ
Conditional compile:XMK_ADD_CBREAK_LOGIC
```

Message description: This message is send if the user typed in the appropriate command.

- Message tag: CMD_BREAK_INPUT_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Set breakpoint on SDL input
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_BREAK_INPUT_REQ
Conditional compile:XMK_ADD_CBREAK_LOGIC
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_BREAK_STATE_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Set breakpoint on nextstate
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_BREAK_STATE_REQ
Conditional compile:XMK_ADD_CBREAK_STATE
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_CLEAR_BREAKPOINT_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Clear specified breakpoint from list
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_CLEAR_BREAKPOINT_REQ
Conditional compile:XMK_ADD_CBREAK_LOGIC
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_CLEAR_ALL_BREAKPOINT_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Clear all breakpoints
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CBREAK_LOGIC
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_QUERY_BREAKPOINT_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Query the breakpoint list
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CBREAK_LOGIC
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_QUERY_ERROR_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Query last error occured in system
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CQUERY_ERROR
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_NEXT_STATE_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Modify processstate to value
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_NEXT_STATE_REQ
Conditional compile:XMK_ADD_CNEXTSTATE
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_CREATE_PROCESS_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Create SDL process
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_CREATE_PROCESS_REQ
Conditional compile:XMK_ADD_CCREATE_PROCESS
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_TBUF_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Query Tracebuffer
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CTBUF
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_KILL_PROCESS_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Kill process
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_KILL_PROCESS_REQ
Conditional compile:XMK_USE_CMD_KILL&XMK_ADD_CKILL_PROCESS
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_PCO_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Define PCO
Sent when          :unused
Direction          :H->T
Typedefinition     :-
Conditional compile:-
```

  Message description: This message is currently not implemented and is never sent.

- Message tag: CMD_QUERY_PROCESS_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Query process state
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_QUERY_PROCESS_REQ
Conditional compile:XMK_ADD_CQUERY_PROCESS
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_QUERY_QUEUE_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Query all queues
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CQUERY_QUEUE
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_RMQUEUE_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Flush all queues
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CRM_QUEUE
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_SET_TIMER_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Set SDL timer
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_SET_TIMER_REQ
Conditional compile:XMK_ADD_CSET_TIMER
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_RESET_TIMER_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Reset SDL timer
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_RESET_TIMER_REQ
Conditional compile:XMK_ADD_CRESET_TIMER
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_QUERY_TIMER_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Query timertables
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:XMK_ADD_CQUERY_TIMER
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_ACTIVE_TIMER_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Query, if timer active
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_ACTIVE_TIMER_REQ
Conditional compile:XMK_ADD_CACTIVE_TIMER
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_FLUSH_SIGNAL_BY_SID_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Flush signals with given ID
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_FLUSH_SIGNAL_BY_SID_REQ
Conditional compile:XMK_ADD_CFLUSH_SIGNAL_BY_SID
```

  Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_FLUSH_SIGNALS_BY_PID_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Flush signals of process
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_FLUSH_SIGNALS_BY_PID_REQ
Conditional compile:XMK_ADD_CFLUSH_SIGNALS_BY_PID
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_RESET_ALL_TIMERS_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Reset all timers
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_RESET_ALL_TIMERS_REQ
Conditional compile:XMK_ADD_CRESET_ALL_TIMERS
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_SCALE_TIMER_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Scale timers
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_SCALE_TIMER_REQ
Conditional compile:XMK_ADD_CSCALE_TIMER
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_START_SDL_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Starts the SDL system
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_START_SDL_REQ
Conditional compile:-
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_OUTPUT_TO_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Sends a signal from the environment
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_OUTPUT_TO_REQ
Conditional compile:-
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_OUTPUT_INTERNAL_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Sends an internal signal
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:-
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_GET_CONFIG_REQ

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Gets the target's configuration
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:-
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_REC_OFF

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Switch recorder off
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:-
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_REC_PLAY

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Switch recorder to play mode
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:-
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_REC_RECORD

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Switch recorder to record mode
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:-
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_TRACE_ON

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Switch trace on in general
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:-
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_TRACE_OFF

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Switch trace off in general
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :-
Conditional compile:-
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_TRACE_PROCESS

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Modify traceoptions for process
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_TRACE_PROCESS
Conditional compile:-
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_TRACE_SIGNAL

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Modify traceoptions for signal
Sent when          :User typed in command
Direction          :H->T
Typedefinition     :xmk_T_CMD_TRACE_SIGNAL
Conditional compile:-
```

Message description: This message is sent if the user typed in the appropriate command.

- Message tag: CMD_BREAK_HIT_IND

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Indicate breakpoint hit to host
Sent when          :Breakpoint hit in target
Direction          :T->H
Typedefinition     :xmk_T_CMD_BREAK_HIT_IND
Conditional compile:XMK_ADD_CBREAK_LOGIC
```

  Message description: This message is sent if one of the breakpoints defined in the breaklist has been hit. The SDL system is halted and waits on the next commands from that point on.

- Message tag: CMD_ERROR_CNF

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Negative acknowledge on command
Sent when          :A command couldn't be executed
Direction          :T->H
Typedefinition     :-
Conditional compile:-
```

  Message description: This message is sent if the target was not able to respond to a command from the user.

- Message tag: CMD_TBUF_CNF

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Indicate Tracebuffercontents
Sent when          :Response to user command
Direction          :T->H
Typedefinition     :-
Conditional compile:-
```

  Message description: This message is currently not implemented and will never be sent.

- Message tag: CMD_OKAY_CNF

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Command acknowledge
Sent when          :Command correctly executed
Direction          :T->H
Typedefinition     :-
Conditional compile:unconditional
```

  Message description: This message is sent from target to host as a response for all commands which do not carry parameters of any kind. It is used within the host library as a general confirmation of most messages.

- Message tag: CMD_QUERY_PROCESS_CNF

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Indicate process state
Sent when          :Response to user command
Direction          :T->H
Typedefinition     :xmk_T_CMD_QUERY_PROCESS_CNF
Conditional compile:XMK_ADD_CQUERY_PROCESS
```

  Message description: This message is sent as a reaction to the message CMD_QUERY_PROCESS_REQ. It carries information about the process state.

- Message tag: CMD_QUERY_QUEUE_CNF

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Indicate queue states
Sent when          :Response to user command
Direction          :T->H
Typedefinition     :xmk_T_CMD_QUERY_QUEUE_CNF
Conditional compile:XMK_ADD_CQUERY_QUEUE
```

  Message description: This message is sent as a reaction to the message CMD_QUERY_QUEUE_REQ. It carries information about the queue state.

- Message tag: CMD_QUERY_TIMER_CNF

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Indicate timer table
Sent when          :Response to user command
Direction          :T->H
Typedefinition     :xmk_T_CMD_QUERY_TIMER_CNF
Conditional compile:XMK_ADD_CQUERY_TIMER
```

  Message description: This message is sent as a reaction to the message CMD_QUERY_TIMER_REQ. It carries information about the timer tables.

- Message tag: CMD_ACTIVE_TIMER_CNF

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Indicate, if timer is active
Sent when          :Response to user command
Direction          :T->H
Typedefinition     :xmk_T_CMD_ACTIVE_TIMER_CNF
Conditional compile:XMK_ADD_CACTIVE_TIMER
```

  Message description: This message is sent as a reaction to the message CMD_ACTIVE_PROCESS_REQ. It carries information about the state of a timer according to SDL semantics.

- Message tag: CMD_QUERY_BREAKPOINT_CNF

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Indicate breaklist to host
Sent when          :Response to user command
Direction          :T->H
Typedefinition     :xmk_T_CMD_QUERY_BREAKPOINT_CNF
Conditional compile:-
```

  Message description: This message is sent as a reaction to the command CMD_QUERY_BREAKPOINT_REQ. It carries the breaklist from target to host.

- Message tag: CMD_QUERY_ERROR_CNF

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Indicate last error occured
Sent when          :Response to user command
Direction          :T->H
Typedefinition     :xmk_T_CMD_QUERY_ERROR_CNF
Conditional compile:-
```

  Message description: This message is sent as a reaction to the message CMD_QUERY_ERROR_REQ. It carries information about the last error which has occurred in the target.

- Message tag: CMD_START_SDL_IND

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Indicate ready for configuration
Sent when          :Response to user command
Direction          :T->H
Typedefinition     :-
Conditional compile:-
```

  Message description: This message is sent when the target is ready for configuration from the host. In this state the target's SDL system is not initialized.

- Message tag: CMD_START_SDL_CNF

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Confirm SDL system start
Sent when          :Response to user command
Direction          :T->H
Typedefinition     :-
Conditional compile:-
```

  Message description: This message is sent as a reaction to the message CMD_START_SDL_REQ.

- Message tag: CMD_GET_CONFIG_IND

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :Send target's configuration
Sent when          :Automatic at startup
Direction          :T->H
Typedefinition     :xmk_GET_CONFIG_IND
Conditional compile:-
```

Message description: This is the second message which shows the target configuration. This means the scaling configuration (`ml_mcf.h`) as well as run-time configuration.

- Message tag: CMD_GET_DATA_CONFIG_IND

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :see description below
Sent when          :Automatic at startup
Direction          :T->H
Typedefinition     :xmk_GET_DATA_CONFIG_IND
Conditional compile:XMK_USE_AUTO_MCOD
```

Message description: This is the first message which indicates the target's length, alignment and endians of the basic c-data types.

- Message tag: CMD_GET_CONFIG_CNF

```
Messageclass       :XMK_MICRO_COMMAND
Meaning            :
Sent when          :Response to user command
Direction          :T->H
Typedefinition     :xmk_T_CMD_GET_CONFIG_CNF
Conditional compile:-
```

Message description: This message is the reply message to the host's `CMD_GET_CONFIG_REQ`. It sends the same information as `CMD_GET_CONFIG_IND`.

Used Default Protocol

The data link part of the communications link is represented by the data link module and the coder module. The coder module ensures that each data link frame is encoded / decoded according to definitions of the default protocol.

- The protocol delivered with the Cmicro Package is easy to modify.
- It allows transfer of all the data necessary for each part of the SDL Target Tester
- It allows the transfer of user defined data to be integrated.
- No handshake procedure or retransmission request is implemented

Each frame of the Cmicro Protocol is seen from the physical layer as binary data.

Each frame consists of the following items:

- **`Message Class`**
  Used to classify the information (is it SDL Target Tester or User de-

fined data?). The codes F3h to FFh are free to use by the user, the other ones are reserved for the SDL Target Tester.

- **Message Tag**
  Used to differentiate the different information either of the SDL Target Tester or the user defined data. For user defined data, the message tag is free to use. If more than 255 bytes are to be transferred, the most significant bit of the message tag is set, otherwise it is cleared. This means there are up to 128 free values for the message tag.

- **SYNC**
  Fixed marker 0xaa is used to check if the receiver is synchronized with the sender. This is to detect problems in the start-up phase of host and target. This field of the message header is used in combination with the message class (>= F0h) to re-synchronize the data stream in the host's data link layer (sdtgate).

- **Message Length**
  Used to store the information about how many bytes are to follow after the length information, excluding the CRC information. The message length is normally encoded in one octet, which means the following data has a maximum length of 255 octets. If there is more data the most significant bit of the message tag is used to indicate that the message length exceeds 255 octets. In that case the message length is encoded in 2 octets which makes it possible to transfer 65536 octets of data.

- **Data**
  Free formatted data (user data or data for the SDL Target Tester).

- **Checksum**
  Used to check if the frame is OK on the receiving site. The checksum is built beginning from tag1 to the last data byte by a modulo-8 operation. The sync byte is included in the CRC.

The following is an example of a coded Cmicro Protocol frame for a MS DOS PC, where the message tag is coded as `CMD_TSTATIC_CREATE`. Note this is only an example and the real coding may be different.

Example 624: Coding for MS-DOS PC ----------

Message-class	Message-tag	SYNC	Message-length	-any kind of data-	Checksum
0xF1	0x06	0xAA	0x02		0x??

The functions described in "The API on Target" on page 3679 can be used to encode / decode these frames.

The Communications Link's Target Site

The Data Link Layer

General

The data link of the communications link usually handles the separately message transfer of information from host and to host.

The data link uses the services and functions of the physical layer in order to transmit and receive information.

The data link offers functionality that makes it possible to send and to receive messages via the physical layer assigned to the data link. This functionality can be used by all programs (if implemented so). Mainly it is the SDL Target Tester which uses the data link layer services in order to send and to receive messages. But the users may want to send and receive messages, too. This is possible by using the data link layer functions and by regarding the defined protocol used for each message.

The data link layer offers the following functionality:

- Receive a message from any part of the system like kernel SDL processes or user defined C functions.
- Store messages in a buffer.
- If the physical layer is ready with the transfer of the last message, send the next message to the physical layer.
- If the physical layer indicates a complete message, store it into the receiving buffer.

- Deliver the messages, which have been stored within the receiving buffer to the requesting program parts.
- Poll the physical layer.
- Handle both directions in parallel.
- CRC procedures (if necessary).

The data link layer is represented by the following functions, which are exported by the data link (`mg_dl`) module.

Data Link Functions Used by SDL Target Tester

The SDL Target Tester's target library uses the following C functions of the link and buffering file `mg_dl.c`. The body of each C function may be rewritten according to the users needs.

`xmk_DLInit`	Initialization of data link
`xmk_DLQuery`	Polling the receiver of data link
`xmk_DLRead`	Reading a message from data link
`xmk_DLWrite`	Writing a message to data link
`xmk_DLDeinit`	Deinitialization of data link

These functions are fixed into the SDL Target Tester's data link layer and they handle the mapping and the access to the physical layer.

The Physical Layer

General

Any type of physical communications link can be used to connect host and target, e.g. it could be a V.24, a TCP/IP or a IEEE 488 interface or it could be any other serial or parallel interface.

The physical layer of the communications link normally handles the byte-wise transfer of messages from host to target and the other way around.

It uses the services of the underlaying hardware like interrupts, CPU registers or memory mapped I/O.

The physical layer offers functionality to the data link layer in order to make it possible to transfer any kind of information, including SDL Target Tester data and / or user data.

The physical layer offers the following functionality:

- Receive a buffer which is to be transferred from the data link layer.
- Send this buffer byte-wise via special hardware (or OS - system call) to the receiving system (either host or target).
- If all characters of this buffer have been transmitted, request next buffer.
- Receive characters from the sending system via special hardware (or OS - system call).
- Collect characters until a message is complete; characters are stored within a temporary buffer.
- If message is complete in temporary buffer, transfer it to the data link layer.
- Handle both directions simultaneously.
- CRC procedures (if necessary).

Physical Layer Functions Used by the Data Link Layer (Default)

As described above the data link layer of the SDL Target Tester's target library is shown in the files `mt_cod.c` and `mg_dl.c`. Please note that `mg_dl.c` is only a template with a default implementation for the connection between data link and physical layer.

Distributed with the SDL Target Tester's target library, there are several template files for the physical layer (TCP/IP and V.24 interfaces) contained in the `cmicro/template/commlink` directory in the installation directory. These template files are tested only for the specific hardware in the Cmicro Package test environments. An error free compilation or execution cannot be guaranteed for the user's target hardware.

There are physical layer templates for the following target hardware:

- Intel 8051 derivatives -> 8051_v24.[ch]
  Franklin/Keil C51 and IAR C51 compilers

- Siemens 80C166 -> 80166v24.[ch]
  BSO/Tasking 80C166 compiler
- MS Windows NT/95 -> win32v24.[ch]
  Microsoft VC++ 6.00
- MS Windows NT/95 -> win_sock.[ch]
  Microsoft VC++ 6.00
- SunOS 5.x -> ux_sock.[ch]
  GNU gcc, Sun WorkShop compilers CC and cc

For use in the target executable, one of the modules listed above is automatically included in `mg_dl.c`. Which physical layer module is included depends on the compiler selected and the flags set in `ml_mcf.h` (e.g. `XMK_USE_V24 or XMK_USE_SOCKETS`).

The template module `mg_dl.c` works as follows:

The function `xmk_DLInit` initializes the buffers used for the data link and calls the initialization of the physical layer (by default this is `xmk_V24Init`).

`xmk_DLDeinit` is the counterpart of `xmk_DLInit` and frees the interface device. This function is normally empty in microcontroller applications but has to be used on PCs, under Microsoft Windows or UNIX.

`xmk_DLWrite` is called when a complete frame should be sent to the host. There are two ways the connection to the physical layer can be handled.

1. Blocked

   If a frame is to be sent it is necessary that the previous frame has already been completely sent. If not, the data link has to wait until the previous frame is sent. See the module `mg_dl.c` (for the compilers KEIL_C51 and IARC51) to get an idea of this connection.

2. Unblocked

   If the frame should be sent, it is handed over to a ring buffer. Although the previous frame has not been sent completely, SDL can execute because the data link does not have to wait.

   One exception must be mentioned: If the transmitter's ring buffer (used for V.24 communication) is full (macro

XMK_MAX_SEND_ENTRIES) the data link will still have to wait here. Also the size allowed for the ring buffer's entries must be great enough (to verify this see the macro XMK_MAX_SEND_ONE_ENTRY.

xmk_DLRead will be polled with each cycle of the SDL loop xmk_RunSDL (see module mk_main.c). If the transmitter and the receiver of the physical layer are not implemented as interrupt service routines (which is strongly recommended) the transmit and receive functions have to be called here (xmk_V24Receive and xmk_V24Send for the distributed V.24 templates). In addition for the V.24 interface the XON handshake has to be called here doing xmk_V24Handshake.

As in xmk_DLWrite there are two ways the receiver buffers can be handled in xmk_DLRead. It is also possible to use a ring buffer for the receiver's side. Similar as in xmk_DLWrite it is necessary to fill the macros XMK_MAX_RECEIVE_ONE_ENTRY and XMK_MAX_RECEIVE_ENTRIES with useful values.

Steps to Implement a Communications Link

Extension to the Template Makefile

In the distribution of the Cmicro Package there are several template makefiles.

On UNIX the existing makefile templates are called makeoptions.no_tester and makeoptions.with_tester.

In Windows the existing makefile templates are called makeno_t.opt and makew_t.opt.

The intention is to give one template makefile where the SDL Target Tester is not included (which means that there is no compilation of any SDL Target Tester C code), and to give one template makefile including the SDL Target Tester.

In the template makefiles that include the SDL Target Tester, it is necessary to define the communications link software by adding the name of the C module(s) that contain(s) this. This can be achieved by changing the list of the object files given in sctLINKTARGET_WITH_TESTER (e.g. 8051_v24.suffix). In addition, the compilation rules for this/these additional module(s) must be specified as well in the makefile.

Finally, the makefile name must be changed into `makeoptions` on **UNIX**, and into `make.opt` in **Windows**.

> **Note:**
>
> The `makeoptions` and `make.opt` files below the predefined kernel directories in `<sdtdir>/SCMA*` are not intended to be used for target or target debug applications.

Adapting the Data Link Layer

The data link layer for the SDL Target Tester on target site is represented by the module `dl.c` with the functions

- `xmk_DLInit()`
- `xmk_DLRead()`
- `xmk_DLWrite()`
- `xmk_DLQuery()`
- `xmk_DLDeinit()`

Several ways to fill these functions are already given in `mg_dl.c`.

The data link layer can be build using a ring buffer for all the trace message or writing each message directly to the physical layer.

Furthermore the data link layer must be build to work together with the physical layers (E.g. ISR driven or not).

It is recommended to have a look into the distributed template `mg_dl.c` which shows different ways.

- For using the ring buffer, please follow the flag `TCC80166`.
- For using no ring buffer, please follow the flags `IARC51` or `KEIL_C51`.
- For using an ISR driven physical layer, please follow the flags `IARC51` and `KEIL_C51`.
- For using a not ISR driven physical layer, please follow the flag `TCC80166`.

Adapting the Physical Layer

There are given tested physical layers for the micro controllers 8051 and 80166.

The structure of these physical layers should be taken and filled with the functions for the needed communications link. E.g. the function to write to a serial interface must be replaced by a function to write to an Ethernet interface.

The Communications Link's Host Site

The sdtgate is one executable of the SDL Target Tester's tool chain on host site. It is built to simplify the communication between the host and the target.

In this release of the SDL Target Tester, the sdtgate is built using a V.24 interface and the sockgate is build using a TCP/IP interface. But as you can see in this subsection, it will be very simple to implement any other communication interface as a new executable.

Communication between the SDL Target Tester Executables

The SDL Target Tester executables `sdtmtui`, `sdtmt` and `sdtgate` communicate with the help of the Cmicro Postmaster. The Cmicro Postmaster handles all the communication between the SDL Target Tester tool chain and is started automatically when the SDL Target Tester is invoked.

The Default V.24 Connection: Sdtgate

Fork Parameters for the Sdtgate

There are two groups of fork parameters handed over to the sdtgate.

The Sdtgate Device Specifications

Inside the SDL Target Tester configuration file `sdtmt.opt` there are 7 entries for the gateway executable.

Example 625: On UNIX

```
USE_GATE              <pathname>/sdtgate
GATE_CHAR_PARAM_1     /dev/ttya
GATE_CHAR_PARAM_2     0
GATE_CHAR_PARAM_3     0
GATE_INT_PARAM_1      9600
GATE_INT_PARAM_2      0
GATE_INT_PARAM_3      0
```

On UNIX, it muse be ensured that the device (`/dev/ttya` above) is accessible. Ask the system administrator if there are problems to open this device.

Example 626: In Windows

```
USE_GATE              <pathname>/sdtgate.exe
GATE_CHAR_PARAM_1     COM2
GATE_CHAR_PARAM_2     0
GATE_CHAR_PARAM_3     0
GATE_INT_PARAM_1      9600
GATE_INT_PARAM_2      0
GATE_INT_PARAM_3      0
```

USE_GATE gives the name of the used gate only.

Inside the delivered sdtgate for V.24 interface only the GATE_CHAR_PARAM_1 (a string) and the GATE_INT_PARAM_1 (an integer) are used. For future developments there are 2 further string parameters and two further integer parameters which are handed over when forking the gate.

The delivered version of the sdtgate uses fixed values for parity, start bits, stop bits...

These are: databits = 8, startbits = 1, stopbits = 1, parity = none

Coding Rules

There are two coding rule pieces of information the gate needs to work.

These are the length of a character on the target in octets (normally 1) and the position of the character. This value is necessary for special micro controller memory layouts e.g. the MSP58C80 gets the character length of 2 octets but the used character is the lower octet only. In this case the character position is 1. Normally if the character size is 1 the character position is 0.

The TCP/IP Connection: Sockgate

The Sockgate Device Specifications

Like the sdtgate, the sockgate is using only 3 of the 7 entries for the gateway executable inside the SDL Target Tester configuration file `sdtmt.opt`.

Example 627: ----------

```
USE_GATE              <pathname>/sockgate
GATE_CHAR_PARAM_1     207.46.138.0
GATE_CHAR_PARAM_2     0
GATE_CHAR_PARAM_3     0
GATE_INT_PARAM_1      9000
GATE_INT_PARAM_2      0
GATE_INT_PARAM_3      0
```

USE_GATE gives the name of the used gate only.

It must be ensured that the IP-address (`207.46.138.0` above) is accessible.

Inside the delivered sockgate only the `GATE_CHAR_PARAM_1` (a string) and the `GATE_INT_PARAM_1` (an integer) are used. The delivered version of the sockgate uses an address string (a valid IP-address) and a port number (valid between 1 and 65535).

Building an Custom Made Communications Link

All the C files and libraries necessary to re-build the default gateway sdtgate are delivered with the SDL Target Tester. The files for a custom built gateway are named usergate to distinguish between the custom and the delivered sdt- and sockgate.

The Modules Building a Usergate

The following files are part of the usergate:

```
cmicro
   + sdtmpm
   |    + sunos5
   |    |    + sdtmlib.a (SunOS 5 version)
   |    + wini386
   |         + sdtmpmcl.lib (Microsoft C++ 6.0 version)
   |         + sdtmpmcl5.lib (Microsoft C++ 5.0 version)
   + mcod
   |    + mcodfnc.[ch]pp
   |    + mcodlib.[ch]pp
   |    + mcod.h
   + sdtgate
   |    + mg_ctrl.[ch]
   |    + usergate.rc (Windows only)
   |    + sunos5
   |    |    + usergate.[ch] (SunOS 5 version)
   |    + wini386
   |         + usergate.[ch] (Windows version)
   + template
        + mg_dl.[ch]
        + commlink
             + <interface>.[ch]
```

Files with suffix `.h` and `.hpp` are not listed but are still being used.

<interface> stands for the selected interface type.

On UNIX the module `ux_v24.c` is used (`ux_sock.c` for sockets).

In Windows the module `win32v24.c` is used (`win_sock.c` for sockets).

The purpose of each module is as follows:

- `mcodfnc.cpp`

  This module delivers all functions to decode and encode the messages sent between the different executables of the SDL Target Tester tool chain. All functions are contained within a C++ class called ClassHostMessage.

  As these functions have to be suitable for the executables sdtmt and sdtmtui, no changes need be made by the user.

- mcodlib.cpp

  This module is automatically generated by the Message Coding Generator (MCOD) and delivers a decode or encode strategy for each message that can be exchanged between the parts of the SDL Target Tester tool chain.

  No modifications can be done inside this module.

- mcod.h

Like `mcodlib.cpp` this file is generated by MCOD too. It contains all MessageTags for the communication between sdtmtui, sdtmt and sdtgate and no modifications are to be done here.

- mg_ctrl.c

  In this module the reactions of the messages exchanged between the SDL Target Tester tool chain are invoked. Like `mcodlib.cpp` and `mcodfnc.cpp` there is no need to make any modification inside this module because the communication between the different parts of the tool chain depends on it.

- usergate.rc **(Windows only)**

  This resource file is for use in MS Windows executables only. It just needs to be compiled and to be linked together with the other object files.

- usergate.c

  This file handles all the initialization and communication of the sdtgate with the Cmicro Postmaster.

  This file is platform specific and needs to be modified in some cases, e.g. when using the V.24 or socket interface on MS Windows a Windows Message is generated when a character is received or completely sent. The reaction on the WM_COMMNOTIFY message can be removed here if another interface is used.

- sdtmlib.a/sdtmpmbc.lib/sdtmpmcl.lib

  This library presents the connection to the Cmicro Postmaster.

- <Interface>.c

  The last of the listed files needs some modifications if the communication interface is to be changed.

  There are eight functions inside this module

  – `xmk_V24Init`() and `xmk_SocketInit()`
  These functions are called only once after the sdtgate has been forked. All initialization of the communication device have to be done here.

Errors should be given as plain text and will be sent to the user interface.

- `xmk_V24Deinit()` and `xmk_SocketDeinit()`
  These functions may not need to be filled for the wanted communication interface. If it has to, all deinitialization can be done here. This function is called directly before exiting the gate.

- `xmk_V24Handshake()`
  The V.24 RS-232 protocol describes the use of the XON - XOFF communication. In the SDL Target Tester's host - target communication XON characters sent only every second. If using another communication protocol (like RS-422/RS-485) this function must be left empty.

- **`xmk_V24Receive()`** and `xmk_SocketReceive()`
  This function does several things.
  At first the communication interface is polled and all available characters are taken from the interface. Then the gate checks the received character stream for the correct sync field position, the length field is read and the amount of characters used to build one complete target message is sent to sdtmt, unless there are no complete messages left.
  The re-synchronization of the data stream is also done in this function if the sync field is not in the correct position.

- **`xmk_V24Send()`** and `xmk_SocketSend()`
  If the gate gets a message from sdtmt which is to be transmitted to the target, this function is called.
  There are several ways to transmit the whole message. For instance, on Windows NT it is possible to hand over a character buffer to the serial interface. The rest of the transmission is being done by the interface itself.

- `xmk_ExtractControl()` / `xmk_MaskControl()`
  These two functions are for use with an XON-XOFF protocol. If a message contains an XON or XOFF character, in front of this XON or XOFF an ESC is inserted in `xmk_MaskControl()` to differ this (XON/XOFF-) character from the control XON/XOFF.
  On the other hand if a message containing XON/XOFF characters is received, the function `xmk_ExtractControl()` removes all ESC from the data stream.

These functions are called from `xmk_V24Send()` and `xmk_V24Receive()` only and the function calls should be removed if the new communication interface does not support the XON-XOFF protocol.

Compiler Settings

- **On UNIX:**

  The following defines have to be made. The user can use the makefile `makefile.template` in the directory `$sdtdir/cmicro/sdtgate` as a template to compile and link his own gateway.

  - `_GCC_`
  - `XMK_UNIX`
  - `SDTGATE`
  - `XMK_USE_V24` (for the default gateway sdtgate)
  - `XMK_USE_SOCKETS` (for the socket gateway)

- **In Windows:**

  The source code for the sdtgate is tested only in combination with the Microsoft C++ compiler 6.x on a Microsoft Windows installation.

  The following defines have to be made.

  - `MICROSOFT_C`
  - `XMK_WINDOWS`
  - `SDTGATE`
  - `XMK_USE_V24` (for the default gateway sdtgate)
  - `XMK_USE_SOCKETS` (for the socket gateway)

Current Restrictions for Communications Link Software

SDL Restrictions

It is impossible to transfer pointer values via the communications link. An example may be the SDL predefined sort charstring which is by default implemented as a pointer in C.

To avoid problems, users must not use SDL sorts that are implemented as pointers (like charstring), whenever it is required to send / receive them via the communications link (e.g. Cmicro Tracer or Cmicro Re-

corder functionality). This means, that this kind of sorts should only be used in signals inside the SDL system but not at it's environment.

Message Length

The maximum length of the message length is restricted.

An explanation follows:

Each message is transferred as:

```
Header + Data + CRC
```

The length field contained in the header may represent the values from 0 to 255. If the MSB in the message tag is set to "1", then the length field is expected to be a 2 octet values, which makes it possible to represent the value from 256 to 65535.

However, the length field consisting of 2 octets is currently not implemented.

In fact, no message may be greater than 4 + 255 + 1 = 260 characters in total (1 character normally is equal to 1 Byte, which consists of 8 Bits).

Connection of a Newly Implemented Communications Link

The SDL Target Tester uses only a few C function calls in order to send and receive information via the communications link. From the C syntax point of view, users only have to implement the body of these functions and to link them together with the SDL Target Tester's target library into one executable program.

However, when considering timing constraints, critical paths, interrupts and similar things, users have to note the following.

Critical Paths

Critical paths occur in any type of software which uses interrupts. Especially the transfer of information between ordinary C code and interrupt service routines must be considered, e.g. for the communication from the physical layer (which may be implemented as an ISR) to the data link layer (which may be implemented as ordinary C functions).

Critical paths may occur at any place in the software which accesses global C variables.

The beginning of each critical path must be blocked with something like “disable interrupts” (see `XMK_BEGIN_CRITICAL_PATH` in the compiler section of `ml_typ.h`). The end of a critical path must be released with something like “enable interrupts” (see `XMK_END_CRITICAL_PATH`).

Timing Constraints

Often there are timing constraints to be considered within the target system. Timing problems must be prevented. Each ISR must be implemented with a few statements only, in order to make the execution time as short as possible.

Interrupts

C functions which should be compiled as ISRs by the C compiler in use are mostly

marked by a preprocessor directive or by a special C compiler keyword. The C compilers reference manual must be consulted in order to implement ISRs.

Open Interface

The open interface can be used to implement user host executables which are able to communicate with the SDL Target Tester at the target site. The SDL Target Tester at the target site is represented by the target library.

Using the Open Interface

In order to use the open interface in this way, the following are of interest:

- Which messages can be sent to the target. This is described within the earlier subsection “Message Formats” on page 3690.
- Which messages are to be expected from the target. This is also described within “Message Formats” on page 3690.
- The format of these messages. Also described within the earlier subsection “Message Formats” on page 3690.

- The situations in which the messages may be sent. Please refer to the subsection "State Machine and Handshake" on page 3689.
- The situations in which a particular message may be received. Again, please refer to "State Machine and Handshake" on page 3689.

It is not of great interest what the format of the data link frames is because it is possible either to use the existing protocol or to re-implement the protocol (e.g. it may be necessary in order to integrate the SDL Target Tester into an existing communication link's software).

The host site must send each message in target representation.

For instance, if the target system uses integers as 2 octet values with the ordering higher octet first, then the host must send an integer value with higher octet and then lower octet, irrespective of its own layout.

For the other direction (reception of messages at host site), the host must also map the information according to the relation between the target memory layout and its own internal representation.

Accessing the Data Link by User

The user can send information via the communication link by calling the C function `xmk_DLWrite` with appropriate parameters. The following example shows one possibility.

Example 628 ----------

```
#define HelloWorldTag <AnyValueAllowed>

char buffer     [20];
char helloworld [15];
int  length     = 0;
int  write_result ;
strcpy (helloworld, 'Hello World');


buffer [0] = 0x11;
length ++;
/* any value for Messageclass, except      */
/* reserved values for SDL Target Teste    */
/* reserved values begin from 250 to 255   */

buffer [1] = HelloWorldTag;
length ++;
/* any value for Messagetag                */
/* Messagetag is made unique together      */
/* with Messageclass, so that no           */
/* conflict occurs.                        */

buffer [2] = strlen (helloworld);
length ++;
/* length of data for this message         */
/* length is given without calculating     */
/* the CRC field                           */

memcpy (buffer [3], helloworld, strlen (helloworld);
length += strlen (helloworld);
/* It is assumed, that no stringterminator */
/* is included in the message (the length  */
/* can be calculated from Messagelength)   */

buffer [length] = xmk_EvalChkSum (buffer, length);
/* Evaluate the checksum of current buffer */

write_result = xmk_DLWrite (buffer, 5 + strlen (helloworld));
/* transmit buffer via communications link */
```

The user can receive information from the communication link by calling the C function `xmk_DLRead` with appropriate parameters.

Example 629: `xmk_RunSDL ()` ----------

The following section is partially implemented within the module `mk_main.c`

```
p_data = xmk_DLRead () ;
if (p_data != (char *) NULL)
{
  /* if anything received, decode it    */
  /* according to Cmicro Protocol       */
  result = xmk_Cod_Decode (p_data,
                           &MessageClass,
                           &Messagetag,
                           &MessageLength,
                           buffer,
                           sizeof (buffer));
if (MessageClass == XMK_MICRO_COMMAND)
{
  result = xmk_HandleCmd ( Messagetag, (xmk_U_CMDS*) buffer );
  return;
}

/*
** Received user message
*/
if (MessageClass != 0xF8)
{
  ErrorHandler (-7777);
  return ;
}

if (HelloWorldTag == Messagetag)
{
  /* HelloWorldTag is used in this example to */
  /* identify the hello world message         */
  /* The user can choose any value for the    */
  /* Message tag                               */
  char helloworld [15];

  memset (helloworld, 0, sizeof (helloworld));
  memcpy (helloworld, buffer, MessageLength);
  printf ('%s', helloworld);
}
```

The above examples assume that the Cmicro protocol (which is described in "Used Default Protocol" on page 3710) is used.

More Technical Descriptions

The File Formats of Sdtmt

General

This section deals with the general file format of the different files which are written to or read by the SDL Target Tester's tool chain. The section is not of general interest, and need be read only when implementing and connecting other tools to the host. The format of the file containing symbols is described in the subsection "The Symbol Tables" on page 3730.

<infile> and <outfile>

Physically, there is the same file format behind the input files and output files written by sdtmt. These files store the information as received via the communications interface in a 1:1 format. The trace of the SDL execution flow and the information from the record mode is simply copied into the file on the hard disk. The same header information is used as on the communications link. This has some advantages.

Without exception the format of the information stored in <infile> or <outfile> is exactly the same as the format of the Cmicro Protocol. Therefore these files are using the memory layout of the target system.

The reason for this is that binary files are always more compact than readable ASCII files.

The message decoder of the host must be active only when information is to be converted, for example if they are to be displayed. This can decrease the performance a little.

The Symbol Tables

The symbol tables which are used by the SDL Target Tester are automatically generated by the Cmicro SDL to C Compiler. Most of the intelligence to interpret SDL traces and recorded events is implemented on the host system. Normally, the host system has enough memory and performance to do the interpretation, whereas the target has not. This is the reason, why the symbol tables used by the Cmicro Package mainly reside on the host. However, some parts of the symbol table reside

where the Cmicro Kernel physically is executed, i.e. within the target system.

The Target Symbol Table

The following table is generated by the Cmicro SDL to C Compiler. For each SDL process, there is one entry in the table representing a pointer to the different trace options for that process.

```
/*****************************************************
**  Symbol trace table
*****************************************************/
#ifdef XMK_ADD_TEST_OPTIONS
XSYMTRACETBL *xSYMTRACETBL[MAX_SDL_PROCESS_TYPES+1] =
{
  (XSYMTRACETBL_ENTRY *)  NULL,
...........
...........
  (XSYMTRACETBL_ENTRY *)  NULL,
  X_END_SYMTRACE_TABLE
};
#endif
```

The Host Symbol Table

Generated File <systemname>.sym

The host symbol table is generated by the Cmicro SDL to C Compiler into a file called `<systemname>.sym`. This file is to be specified when the host site is invoked (see subsection "Invoking the SDL Target Tester's Host" on page 3619).

In the following, the file format is described with EBNF like syntax:

```
SymbolFile          ::= Header SymbolDescriptions
Header              ::= HeaderComment Timestamp
HeaderComment       ::= "Cmicro .sym file, version x.y"
Timestamp           ::= "TIMESTAMP <GenerationTime>"
SymbolDescriptions  ::= ( DescriptionLine ) ...
DescriptionLine     ::= Depth Subject SDTReference
Depth               ::= [0-9]+
Subject             ::=  SYSTEM IDs
                       | BLOCK IDs
                       | SIGNAL IDs
                       | PROCESS IDs
                       | START IDs
                       | INPUT IDs
                       | OUTPUT IDs
                       | NEXTSTATE IDs
                       | <to be continued>
SDTReference        ::= "#SDTREF(<sdt-reference>)" LF
LF                  ::= "\n"
```

In the following an example is given that was produced with a simple "ping pong" SDL system:

Example 630 ----------

```
Cmicro .sym file, version 2.0
TIMESTAMP <GenerationTime>

1 SYSTEM <SystemName>  #SDTREF(sdtref)
2 BLOCK <BlockName>  #SDTREF(sdtref)
3 PROCESS <ProcessName> 0 1  #SDTREF(sdtref)
4 START 0  #SDTREF(sdtref)
4 INPUT 1  #SDTREF(sdtref)
4 INPUT 2  #SDTREF(sdtref)
4 OUTPUT 3  #SDTREF(sdtref)
4 NEXTSTATE State1 1 4  #SDTREF(sdtref)
4 OUTPUT 5  #SDTREF(sdtref)
4 NEXTSTATE - 0 6  #SDTREF(sdtref)
4 TASK 7  #SDTREF(sdtref)
4 STOP 8  #SDTREF(sdtref)
3 PROCESS <ProcessName> 1 1  #SDTREF(sdtref)
4 START 9  #SDTREF(sdtref)
4 INPUT 10  #SDTREF(sdtref)
4 ASSIGNMENT 11  #SDTREF(sdtref)
4 NEXTSTATE State1 1 12  #SDTREF(sdtref)
4 ASSIGNMENT 13  #SDTREF(sdtref)
4 DECISION 14  #SDTREF(sdtref)
4 OUTPUT 15  #SDTREF(sdtref)
4 STOP 16  #SDTREF(sdtref)
4 OUTPUT 17  #SDTREF(sdtref)
4 NEXTSTATE - 0 18  #SDTREF(sdtref)
4 DCL counter Integer  #SDTREF(sdtref)
3 SIGNAL PING 1  #SDTREF(sdtref)
3 SIGNAL PONG 2  #SDTREF(sdtref)
3 SIGNAL PONG_SAY_BYE_BYE 3  #SDTREF(sdtref)
3 SIGNALROUTE SR1  #SDTREF(sdtref)
3 SIGNALROUTE SR2  #SDTREF(sdtref)
```

The first line, `Cmicro .sym file, version x.y`, gives some comment about the Cmicro symbol file version.

The `TIMESTAMP <GenerationTime>` value is used internally for consistency checks with the SDL Target Tester. With this time-stamp, a rough consistency check for the generated files is performed and warning messages are printed out in the case of an inconsistency.

The lines following the time-stamp information, contain in the first position an integer value indicating the depth of the structuring level in which the information is found in SDL. For example, the SDL system is in level 1, a block in the SDL system level is in level 2, and so on.

The next column in these lines describe the subject, e.g. if it is a system, block, channel, signal route, process, procedure, signal, start, input or output, or whatever.

For each subject there are IDs generated, these are used in the target executable program for generating trace information. The IDs begin with

0 (in the case above it is the start symbol of the first process) and end with the number of the last SDL symbol in the system.

The last column in the lines contains a `#SDTREF` reference, which points to the location of the subject in either SDL/GR or SDL/PR.

Internal Symbol Table Structure

The symbols of a trace can be found during the conversion of internal information into a displayable format.

The following three SDL objects are treated:

- Processes
- Signals/Timers
- States

They are handled in the same way. Namely by building a linked list of symbols for processes, a linked list for signals and a linked list for states.

Cmicro Recorder

The information given in the following sections are valid both for host and target and there is no differentiation made between these.

> **Note:**
>
> The SDL Target Tester's Record and Play functions are only available if a Cmicro Recorder license is available.

Type and Amount of Stored Information

As already mentioned in earlier sections, using the Cmicro Recorder makes more sense in those cases where an SDL system is executed in real-time in the target.

That's why the amount of stored information is to be kept very small. Of course, this leads to the problem that not all types of errors can be found with the Cmicro Recorder but there is a good chance to find most of them. The information which the Cmicro Recorder produces at target site and which the Cmicro Recorder uses at host site cannot be configured in their dimension. The packets which are to be transmitted via the communications link are always the same. A technical description follows in the next section.

Record Mode

First, the SDL system executes the start transitions of all statically created processes. At the occurrence of a “counted” symbol, an internal counter variable is incremented (which was set to zero before going into the SDL start-up phase). The counter will simply be incremented until the first environment signal is sent from the environment to a process in the SDL system, or until the first timer expires within the SDL system.

These are special events for the Cmicro Recorder.

In that case, the Cmicro Recorder on the target site sends the following messages to the host:

```
Host <-------- <Counter value> ------------ Target
Host <-------- <Event> --------------------- Target
```

Both messages are stored in the <outfile> on the host.

The counter is reset to zero again and the procedure restarts by counting the “counted” symbols and sending the above messages to the host, either when an environment signal is sent from the environment to a process in the SDL system, or when the first timer expires within the SDL system.

The <outfile> is closed when the user terminates the record mode on the host site, i.e. it is not possible to close the file by ending the record mode on the target site. The record mode can be finished at any time.

After the record mode has been switched off, it is not allowed to switch the record mode or the play mode on again.

The <outfile> on the host now contains messages like this:

```
<trace event 1>
....
<trace event N>
<Counter value>
<Event>
<trace event N+1>
....
<trace event N+1+M>
<Counter value>
<Event>
```

As can be seen, the following sections are stored in the file whereas each section is optional:

```
For all events in the target
            [Store trace section]
            [Store Counter value section]
            [Store Event section]
```

Counted Symbols

The occurrence of the following SDL symbols will be “counted” by an internal counter (a C integer variable):

STATE	The SDL nextstate operation
INPUT	The SDL input operation
SAVE	The SDL save operation
CREATE	The SDL create operation
STOP	The SDL stop operation
STATIC CREATE	Used at system start, appears for each statically created process
DYNAMIC CREATE	Used at start of a dynamically created process
DECISION	SDL decision
TASK	SDL task (not C Code)
OUTPUT	The SDL output operation (see remarks below)
PROCEDURE	SDL procedure call
SET	SDL action: Set timer
RESET	SDL action: Reset timer
DISCARD	SDL Discard
IMPLICIT CONSUMPTION -	SDL implicit transition

A special case is the SDL output operation:

If the output is from one process in the SDL system to another process in the SDL system or if the output is from one SDL process in the SDL system to the environment, the occurrence of this symbol is counted.

Otherwise if the output is from the environment to a process within the SDL system, the event is relevant for the Cmicro Recorder (as the timer expiry is).

C code in an SDL task is normally not counted when recording. The users are free to count C code by calling:

`xmk_Record (CMD_TTASK)` for the record mode and

C code written into an SDL Task (i.e. by using the #CODE directive)

`xmk_Play (CMD_TTASK)` for the play mode.

Play Mode

In play mode, the target accepts signals coming from the SDL environment only from the Cmicro Recorder. These signals are read from the file which was previously written to in record mode.

Caution!

No environment signals may be handled by the user in the C function `xInEnv`! At the beginning of `xInEnv`, there should be a statement like

```
if (xmk_RecorderMode == XMK_RECORDER_PLAY)
   return (XMK_OKAY));
```

In addition, no signal may be sent by any other environment function (like ISRs) to the SDL system.

At the beginning of the play mode, the host reads the first <counter value> from the <infile> and sends it to the target.

```
Host -------- <Counter value> ---------------> Target
```

The meaning of this is: Please execute the amount of “counted” symbols as indicated in the <counter value>. The target executes the start transitions of all statically created processes (start-up-phase). When preemption is used, it is possible that a transition which is not a start transition

is executed. Each “counted” symbol increments a counter value which is compared against the debit, which was indicated by <counter value>.

If the current counter value reaches the debit counter value, then an environment signal (or timer expiry) plus the next <counter value> is requested by the Cmicro Recorder.

```
Host <------- <Request> --------------------- Target
Host ---------------- <Event> --------------> Target
Host -------- <Counter value> --------------> Target
```

If the target has requested the next <Event> and <Counter value>, the target is waiting for these messages only and is not able to handle other messages. This is not a restriction because the environment is disabled in play mode in the target.

If the end of file of <infile> has been reached, the host indicates the end of the play mode to the target. <Counter value> and <event> are always stored together in record mode. Therefore the end of file can only be reached after reading an EnvSig or Timer expiry.

```
Host -------- <Play mode off> --------------> Target
```

Note:

After the play mode has been switched off, it is not allowed to switch the record mode or the play mode on again.

General Restrictions on Record and Play

Of course, there are restrictions in the use of the Cmicro Recorder, especially for the play mode. However, there may be situations where the Cmicro Recorder does not help in finding the source of the problem. The user must decide when the Cmicro Recorder can be properly utilized.

Firstly, the user has to ensure the same configurations are used for hardware and software, in order to allow comparison between the results of a recorded session with the ones of the replayed session. This is only a general recommendation. It is possible that he even would like to compare two different hardware configurations, and the Cmicro Recorder should help him finding the differences between these. As a more general recommendation we would say: Compare only those configurations that are definitely comparable.

Secondly, starting and ending the record and the play mode must be as follows. Host and target are to be synchronized.

Restrictions when Starting Record

The target may not transfer information to the host until the host has become active.

Therefore, the SDL Target Tester's host site should be started first and the file to write in should be opened.

If the target is started (e.g. reset-button), it has to wait on sdtmt before transmitting information (see "XMK_WAIT_ON_HOST" on page 3505).

Restrictions when Ending Record

Ending the record session is to be done from the GUI (sdtmtui) by closing the <outfile>. The user can enter any exit commands at any time. He should pay attention to the exact time when he exits. If exit is performed in the middle of a running SDL transition in the target, then the traces following will not be stored! The last information may then be unpredictable.

Restrictions when Starting Replay

The target may not transfer information to the host, until the host has become active. The host may not transfer information to the target, before the target has become active.

Therefore the SDL Target Tester's host site should be started first and the file containing the recorded session should be opened.

If the target is started (e.g. reset-button) it has to wait for the host before transmitting information.

Restrictions when Terminating Replay

In principle, the play mode reaches the end when the last stimuli stored in the recorded file was read and sent to the target. Actually, the play mode should end when all the transitions which follow that stimuli have ended. This is a discrepancy which is solved as follows:

When the target has received the last stimuli from the recorded session, it enters the "recorder off" mode. The problem with this is, that timers are simulated during the play mode but are no longer simulated if changing to "recorder off" mode. Timers will expire as they have been set and it is the real-time that will be used to allow them expire instead of the simulated timer expiry in the Cmicro Recorder.

Real-time Play Mode

The host tries to replay the SDL environment events at the same absolute time values as they occurred during the record session, if so required by the user (real-time-play is switched on).

Internally, a time stamp is stored for each environment signal, which is compared within the target with the current value of SDL now. SDL now and the time stamps in the signals are calculated with the value 0 from system reset on. The target generates the SDL now values in any case.

Of course, the processing power of host and target together must be large enough to handle this. As a rule of thumb, the recorded environment signals may not be sent to the SDL system more often than one every few hundred microseconds. But that - of course - also depends on the communications link in use.

Restrictions when Using Preemption

There are some restrictions when the Cmicro Recorder is used in combination with preemption. The problem is that in a real-time execution, signals coming from the environment (this may be a hardware interface) can cause a preemption of a running SDL process by a new process with a higher priority. The preemption may occur at any time, which means at any machine instruction (see notes below). On the other hand, only SDL symbols are registered by the Cmicro Recorder.

This means that the play mode may produce a different system behavior other than the recorded one.

The only alternative to implement an exact reproduction of the recorded session would be the use of special debugger hardware (not a general solution).

Some Additional Comments

Real-time behavior cannot be expected in play mode. When considering the other direction, from SDL to the environment, it is not absolutely necessary to cut this connection off (C function `xOutEnv`). It may be a good idea to let this connection be as it is in record mode. For example, if there is a display driver in the environment, which is controlled by SDL, it would then be possible to see the reactions on the display in the same way as they have been observed in record mode.

Utility Functions

File Structure

Description of Files

The modules and functions described in this subsection are used to handle first in-first out buffering, ring buffering, message coding and other possibilities.

ml_buf.c

This module contains functions to handle first-in first-out buffers and ring buffers.

It is mainly used internally by the SDL Target Tester to write and read trace information into/from a first-in first-out buffer within the RAM.

These functions are also free for users.

Three functions are implemented representing the low level functions. Each entry has to be of the same size, because the handling is in principle similar to an array. The type of each entry is not relevant. In addition, the caller of the functions contained in this module defines the memory for the trace buffer as well as a control structure representing all the internal data and variables necessary to administrate the trace buffer.

As a result, users are free to have more than 1 instance of this buffer type.

ml_buf.h

This is the header file containing the external definitions of `ml_buf.c`.

Trouble-Shooting

What to Do if the SDL Editor Trace Does Not Work?

The SDL Target Tester gets the SDL Editor references from the generated file `<systemname>.sym` in the known form `#SDTREF( ... )`.

As it is possible to generate an SDL system and to test it on another platform it is possible that the paths contained in each `#SDTREF()` are not up to date if using the SDL Target Tester.

All the paths in `<systemname>.sym` must simply be updated.

What to Do if There Is a Warning of the Information/Message Decoder?

If the message decoder detects an error there will be a warning like the one shown in Example 631.

Example 631: warning of the message decoder ––––––––––––––––––––––––––––––

```
**WARNING: There is an inconsitency in the information decoder.
This message will be suppressed form now on.
Message Class: 241,d
Message Tag  : 5,d
Number of data bytes received=6
Number od data bytes decoded =5

  ***Received buffer (32 octets):
  ***f1 05 aa 06 00 00 00 00 - 01 04 ab ee ee ee ee ee
  ***ee ee ee ee ee ee ee ee - ee ee ee ee ee ee ee ee
```

––

This warning must be interpreted as follows:

- In the Cmicro Package there are the following message classes:
  - Cmicro Tester commands = 0xf0 = 240
  - Cmicro Tester trace = 0xf1 = 241 (see Example 631)
  - Cmicro Recorder = 0xf2 = 242

- The message tag can be found in the file "tmcod.h" on page 3677

  In Example 631 the message tag 5 (decimal) is the message `CMD_TSTATE`. Furthermore there is a structure called `xmk_T_CMD_TSTATE`. This structure looks like

```
typedef struct
{
  xPID        process;
  xmk_T_STATE state;
}
```

- The shown buffer from above can be interpreted as:
  f1 -> Cmicro Tester trace
  05 -> `CMD_TSTATE`
  aa -> sync byte
  06 -> message data length 6 octets.

- The message data contains the values `xPID` and `xmk_T_STATE`. In the Example 631 the `xPID` value is an `unsigned int` and the `xmk_T_STATE` is an `unsigned char`.

Note:

The kind of C basic type the `xPID` value is represented by, depends on the SDL system in use (single instance processes only or multiple instance processes). Please view the typedef of `xPID` in the file `ml_typ.h`.

The size of the C basic type depends on the microcontroller and compiler that is used.

By having a look to the C struct (`xmk_T_CMD_TSTATE`) which is represented by the 6 data octets received a value for `xPID` is found like `0x00000000` as an integer is given with the length 4 octets in this example.

Note:

The kind of C basic type the `xmk_T_STATE` value is represented by, depends on the SDL system in use, please view the flag "XMK_USE_HUGE_TRANSITIONTABLES" on page 3489.

The size of the C basic type depends on the microcontroller and compiler in use.

Having a look to the fifth octet of the message data the value for `xmk_T_STATE` is `0x01`, as the length of character is 1 and the alignment of character is 8. Please view "Preparing the Host to Target Communication" on page 3614.

After all in this example there are 5 bytes (octets) decoded (4+1) and 6 received which leads to the inconsistency in the information decoder.

The user is asked to correct the settings in the file `sdtmt.opt` to support the SDL Target Tester's host site with the correct information about the target's memory layout.

Caution!

The Example 631 is an exception concerning the inconsistency in the information decoder.

The compiler used for this example only knows C structures with a size dividable by two which leads to a message data length of 6.

The value 0x04 in the example only represents a filler octet which does not need to be noticed.

Chapter 67 The SDL Target Tester

Chapter

68

Cextreme Code Generator Reference

The Cextreme Code Generator is a SDL to C compiler with many advanced features. It generates code with a small memory footprint, which often can be made even smaller by applying some of the many optimizations available.

The Cextreme Code Generator is inteded to be used to develop applications on embedded systems. It is of course possible to use it for other types of applications as well, but its features are designed to be used when developing embedded systems.

The Cextreme Code Generator can generate applications in two modes, Light and Threaded. The Light integration creates a single-threaded application to be executed on a standard environment, while the Threaded integration gives the user control over how many threads that should be created, what they should execute and more.

Finally, the Cextreme Code Generator generates code that has few macros and is easy to read and understand.

File Structure

An application generated by the Cextreme Code Generator consists and depends on a number of files.

The generated code reflects the contents and behavior of the system described by the SDL model. This code consists of a number of `.c` and `.h` files that are put in the target directory (or, in some circumstances, in a subdirectory to the target directory).

In addition to the generated code, there is a number of files that supports the building of an application. These files, and how they should be managed is discussed in this section.

Essential files

Files found in target directory

Depending on the code generation options, files with the following suffixes and extensions might be found in the same directory as the generated C files.

- `.m`

  This is the makefile for compiling the set of files generated from the SDL model.

- `_env.tpm`

  This is a template makefile to be used to include other `.c` files that should be compiled and become part of the executable. The Cextreme Code Generator generates this template at the same time it generates a file with the environment functions. You can then modify the template and store it under a new name, so it does not get overwritten, and then specify that the modified file should be used during compilation.

- `.ifc` files

  These files contain all important declarations on the outermost system level. It is used to access objects in the generated C code from external code, for example in the file implementing the environment functions.

- `_env.c`

This file contains templates for the environment functions. The environment functions are used to connect the SDL model to its environment. In many cases the details in the environment functions are filled in by adding a user specified file containing a number of macro definitions. If that is the case, the `_env.c` file can be regenerated and still be valid. In more difficult cases the structure of the generated environment template does not fit and then the template should be copied to a new name and modified. In this case the `_env.tpm` file needs to be modified as well to compile the correct environment file.

- `auto_cfg.h`

  This is a scaling and configuration file generated by the Cextreme Code Generator. It contains information about sizes and about what SDL constructs are used. This information is used to determine the size of some data structures and to exclude data and code not needed for the application.

- `extreme_user_cfg.h`

  This is a configuration file generated by the Targeting Expert. It contains information about the build options used when building the application and generating code.

- `.o` / `.obj`

  After compilation the target directory will also contain object files (`.o` or `.obj` in most cases) and an executable (`.sct` normally).

Files found in kernel directory

A number of files are found in the kernel directory, usually located at: `<installation_dir>/sdlkernels/cextreme/kernel`

- `extreme_kern.c` and `extreme_kern.h`

  These files are the basic implementation of the run-time kernel. The file `extreme_kern.h` is included by all `.c` files.

- `sctpred.c` and `sctpred.h`

  These files implement the support for predefined data types.

Files found in RTOS directory

In a subdirectory named RTOS to the kernel directory, all the supported RTOS (real-time operating system) integrations can be found, each integration in a subdirectory of its own.

- `rtapidef.h` and `rtapidef.c`

Each integration contains two files, `rtapidef.h` and `rtapidef.c`. As of today, the following integrations are supported:

- POSIX pthreads integration in subdirectory POSIX
- Win32 integration in subdirectory Win32
- VxWorks integration in the subdirectory VxWorks

Include structure for C files

extreme_kern.h

The top level for definitions is the `extreme_kern.h` file. This file is included by all `.c` files and includes in turn a number of other `.h` files according to the list below.

This file includes:

- standard C header files like `string.h`, `stdlib.h`, `limits.h` etc.
- `extreme_user_cfg.h`
- `auto_cfg.h`
- `comphdef.h` (compiler/hardware integration)
- `rtapidef.h` (run-time API integration)
- `sctpred.h`

extreme_kern.c

Include is also used for `.c` files. In this way the complete kernel will be compiled when the top element, `extreme_kern.c`, is compiled.

- `extreme_kern.h`
- `rtapidef.c` (run-time API integration) if you select this

- `sctpred.c`

Environment Functions

General

Virtually all real applications have some physical environment, let it be software or hardware. The SDL model describes on a high abstraction level the interaction the environment (sending or receiving signals or remote procedure calls), but not what should really happen. A signal sent from the SDL system should for example cause a hardware register to be set or a TCP/IP packet to be sent over a network. The implementation of the interaction is provided in the environment functions.

Separating the decision that an interaction should occur from its implementation simplifies the understanding of the overall behavior of the model. It is also easier to simulate or validate the model, as in such a situation it is necessary to have full control over the environment. For example, it is usually not suitable to simulate on the target platform, which makes hardware unable to access.

The following environment functions are available for use:

```
extern void xInitEnv (void);
extern void xCloseEnv (void);
extern void xOutEnv (xSignal *);
extern void xInEnv (void);
```

- `xInitEnv` and `xCloseEnv` are used to initialize and close down the environment in a controlled way
- `xOutEnv` is called by the run-time kernel when a signal is sent from the system to the environment and should implement the actions needed when signals are sent from the system.
- `xInEnv` provides the reverse support, i.e. to send a signal into the system due to events in the environment. The details depend somewhat on the situation and will be described below.

The usage of the environment functions is controlled in the Targeting Expert by specifying which environment function should be used. This results in the possible inclusion of the macro defines

```
#define USER_CFG_USE_xInitEnv
#define USER_CFG_USE_xCloseEnv
#define USER_CFG_USE_xOutEnv
#define USER_CFG_USE_xInEnv
```

in the file extreme_user_cfg.h generated by the targeting expert.

xInitEnv

This function is to be used to initialize the environment. The function is called once during the initialization of the application (in function xInit in uml_kern.c).

xCloseEnv

This function can be used to close down the environment. The function is called once in the main function in extreme_kern.c.

Note: In an RTOS integration this might not be a suitable way to perform the closing of RTOS tasks – refer to the technical documentation of the RTOS how to properly close down OS threads.

xOutEnv

The xOutEnv function passes a pointer to a signal that is sent to the environment. It is called from the signal sending functions in extreme_kern.c (xOutput and xOutputSimple). The function should perform whatever action that is necessary when the signal passed as parameter is sent to the environment.

The function should free the memory of any signal parameters that is implemented using pointers. The handling of the signal itself is, however, performed by the calling functions. In the generated template for the xOutEnv function that is discussed below, the proper code for memory management is automatically inserted. An example of an xOutEnv is also shown in the section on the template environment functions.

xInEnv

The xInEnv function is a way to handle signals that are sent from the environment to the application. If it is a suitable way, or not, depends on the actual application and the integration mode. Independent of how and where signals to be sent into the system are handled, there are two functions in extreme_kern.c that should be used:

```
extern xSignal*
  xGetSignal (xSignalId, int, xSignal **);
extern void
  xENVOutput (xSignal *, xuint8, SDL_Pid);
```

First, xGetSignal is used to get a pointer to a signal data area. Then the signal parameters should be assigned their values and last the signal is sent using the xENVOutput function.

xGetSignal parameters:

- first parameter to xGetSignal: The signal identity (a number).
- second parameter to xGetSignal: The size of the data area for parameters. If the system contains no signal, no timer, and no part with parameters, then this parameter is not used at all.
- third parameter to xGetSignal: This is an optimization parameter that should always be 0.

xENVOutput parameters:

- first parameter to xENVOutput: Reference to the signal obtained by xGetSignal
- second parameter to xENVOutput: Signal priority for the signal. If signal priorities are not enabled, then this parameter is excluded.
- third parameter to xENVOutput: The receiver of the signal.

More details on how to give these parameters can be found in the sections below.

Implementing signal sending to the application

There are three typical cases of how to implement signal sending into the application.

Sending without use of xInEnv

In the first case the xInEnv function is not used. Consider the situation of a bare integration (no underlying RTOS). In that case it is possible to send signals directly in interrupt routines into the system, using the functions described above. To protect the data structure for signals and signal queues it is then necessary to implement functions or macros to disable and enable interrupts.

Sending using xInEnv in bare integration

Case two is also a bare integration (no underlying RTOS). However the signals are not sent directly in the interrupt routines. Instead the inter-

rupt routines are just used to set up some global data structure to remember information about external events. The `xInEnv` function is then used to actually send the signal. `xInEnv` will be repeatedly called from the scheduler (function `xMainLoop` in `extreme_kern.c`) between each transition executed by the system. In each call the `xInEnv` function should check for external events and send the appropriate signal(s) into the system. `xInEnv` should the return as fast as possible to the scheduler. It is up to you to protect the data structure used to remember external event. The data structures in the Cextreme Code Generator kernel need not to be protected in this case.

Send signals with xInEnv in a RTOS integration

Case three handles the situation of an RTOS integration. In that case it is suitable to run `xInEnv` in a thread of its own. In the predefined integrations the main thread runs `xInEnv` after it has created the other threads. The `xInEnv` function should in this case look something like:

```
while (1) {
  wait for an event;
  if (event corresponding to signal 1) {
    send signal 1;
  }
  if (event corresponding to signal 2) {
    send signal 2;
  }
  and so on;
}
```

The “wait for an event” should hang this thread when there is nothing to do. Otherwise this thread might run all the time. When something happens that should cause a signal to be sent to the environment, the code where this has been detected should store information in some global data area (protected!) and execute code to restart the `xInEnv` function. In simple cases just a semaphore can be used to handle `xInEnv`.

A polling solution can be obtained in the structure above by just implementing “wait for an event” as a sleep for an appropriate amount of time. During each turn the `xInEnv` will then check if some external event has occurred and send the corresponding signal.

Note: If a polling solution for xInEnv is choosen it is important that the thread is suspended for a long enough time. The minimal time to wait depends on the pace in which injected signals can be consumed by the application. This in turn can depend on various things, such as the design

of the model, which thread deployment that is used, or even on how threads are allocated on different hardware artifacts. In general you must make sure that signals are not injected in the application at a faster pace than they can be consumed. Otherwise the application will eventually run out of resources.

Interface header file (.ifc)

The `.ifc` file is a system interface header file containing information about entities defined inside the system. For such entities code is generated. Some of these definitions can be useful in external code, like for example the envi-ronment functions. In the generated code for the system, except the `.ifc` file, all SDL name are prefixed or suffixed to make the names unique in C. In the `.ifc` file the SDL names have a predefined prefix.

In the `.ifc` file names for the parts in the system can be found. These names can be used as receiver, when sending signals in the `xInEnv` function.

Generated environment functions

In the beginning of the `systemname_env.c` file the following statements can be found.

```
#include "extreme_kern.h"
#ifdef XENV_INC
#include XENV_INC
#endif
#include "exenv.ifc"
```

By defining `XENV_INC` to a file name it is possible to include a user defined file. This macro is suitable to define in the `extreme_user_cfg.h` file generated by The Targeting Expert. Use the possibility to include your own defines and insert something like:

```
#define XENV_INC "my_defines.h"
```

In this way the skeletons for the generated environment functions can be filled in using macro definitions from the included file. The environment functions can then be regenerated without the risk of overwriting manually inserted or modified code.

xInitEnv and xCloseEnv structure

The `xInitEnv` and the `xCloseEnv` functions have the following structure:

```
extern void xInitEnv(void)
{
    /* Code to initialize the environment
       may be inserted here */
    XENV_INIT
}
extern void xCloseEnv(void)
{
    /* Code to bring down the environment in a
controlled
       manner may be inserted here. */
    XENV_CLOSE
}
```

where `XENV_INIT` and `XENV_CLOSE` are empty macros if you have not defined them earlier. These macros should be the sequence of statements and function calls needed at the initialization and termination.

The function `xInitEnv` is surrounded by:

```
#ifdef USER_CFG_USE_xInitEnv
#endif
```

(similar for `xCloseEnv` as for `xInEnv` and `xOutEnv` discussed below) to compile the function only if you have specified that the function should be used. This is set in the `extreme_user_cfg.h` file by the Targeting Expert.

xOutEnv structure

The generated `xOutEnv` function will have the following structure.

```
extern void xOutEnv(xSignal *SignalOut)
{
  OUT_START_CODE
  /* Signal s1 */
  #ifndef OUT_SIGNAL_sig_s1
    #define OUT_SIGNAL_sig_s1
  #endif
  if (SignalOut->Sid == sig_s1) {
    OUT_SIGNAL_sig_s1
    return;
  }
  /* Signal s2 */
  #ifndef OUT_SIGNAL_sig_s2
    #define OUT_SIGNAL_sig_s2(P1, P2)
  #endif
  if (SignalOut->Sid == sig_s2) {
```

```
      OUT_SIGNAL_sig_s2(
          ((ySignalPar_sig_s2 *)SignalOut)->Param1,
          ((ySignalPar_sig_s2 *)SignalOut)->Param2)
      xFreeSignalPara(SignalOut);
      return;
    }
  }
```

where `OUT_START_CODE` is an empty macro if you have not defined it. The code consists of a sequence of if statements, each handling one of the signal types that can be sent to the environment. For each signal the if expression tests the signal identity.

When the correct if statement is found the statements defined by the macro `OUT_SIGNAL_signalname` is executed. This macro has one parameter for each signal parameter and is empty if you have not defined it. This means that the code will compile, but do nothing if the `OUT_SIGNAL` macros are not defined. The structure makes incremental development possible, that is to say that you can implement the treatment of a few signals and then start testing, without bothering about the code for all the other signals.

The code in the example above assumes that the name for signals used in the `.ifc` file is `sig_%n`, where %n is the signal name in SDL. The `xFreeSignalPara` function call in the s2 section is necessary when any of the signal parameters is implemented using dynamic memory and a `free` operation is needed for the parameter value. The data area for the signal itself is handled in `extreme_kern.c` at the places where `xOutEnv` is called.

xInEnv structure

The generated `xInEnv` function has the following structure. The example below shows an example suitable in a bare integration.

```
extern void xInEnv (void)
{
  xSignal *SignalIn;
  IN_START_CODE
  /* Signal s1 */
  #ifdef IN_SIGNAL_sig_s1
  if (IN_SIGNAL_sig_s1) {
    SignalIn = xGetSignal(sig_s1, 0, 0);
    xENVOutput(SignalIn, IN_RECEIVER_s1);
  }
  #endif
  /* Signal s2 */
  #ifdef IN_SIGNAL_sig_s2
  if (IN_SIGNAL_sig_s2) {
```

```
    SignalIn = xGetSignal(sig_s2,
        sizeof(ySignalPar_sig_s2), 0);
    ((ySignalPar_sig_s2 *)SignalOut)->Param1 =
        IN_PARA1_sig_s2;
    yAssF_s_7(
        ((ySignalPar_sig_s2 *)SignalOut)->Param2,
        IN_PARA2_sig_s2,
        XASS_MR_ASS_NF);
    xENVOutput(SignalIn, IN_RECEIVER_s2);
  }
  #endif
  IN_END_CODE
}
```

where the macros `IN_START_CODE` and `IN_END_CODE` are empty if you have not defined them.

Each signal is treated in four steps.

- The enabling macro `IN_SIGNAL_signalname`. This is used in an if statement to determine if an external event has occurred that should cause the signal to be sent into the system.
- The `SignalIn` variable is assigned a new signal data area.
- The signal parameters are filled in, if any. The value for each parameter should be defined using the appropriate macro `IN_PARA1_signalname`, `IN_PARA2_signalname` and so on.
- The signal is sent by calling `xENVOutput`. Here the receiver of the signal must be given by the macro `IN_RECEIVER_signalname`. The appropriate values can be found in the `.ifc` file looking for defines of type:

```
#define xPartNo_Partname  <integer number>
```

The structure above also enables incremental development as the complete section for a certain signal is removed if the enabling macro is not defined.

For a threaded application the structure is very similar. A loop is included in the function according to the following.

```
extern void xInEnv (void) {
  xSignal *SignalIn;
  IN_START_CODE
  while (1) {
    IN_WAIT_FOR_ACTION
    /* To avoid that the thread running xInEnv takes
       all resources it should wait on for example a
       semaphore until something occurs that should
```

```
            cause a signal to be sent into the system */

          /* Here the code for the signals is placed
             in the same way as in the previous example.
          */
        }
        IN_END_CODE
      }
```

Note: IN_WAIT_FOR_ACTION must be implemented according to the previous discussion.

Compile and Link an Application

Integration types

Cextreme comes with two different kind of integrations, Light and Threaded.

- Light Integration

  The Light Integration is a single threaded integration that requires very little interaction with the underlying OS. The SDL system runs as a single OS task, scheduling is handled by the standard kernel running a complete state transition at a time. The worst case scheduling latency is that of the longest transition.

- Threaded Integration

  The threaded integration can use multiple threads where the scheduling and communication in each thread is controlled by the SDL kernel, but scheduling and synchronization of data (such as input queues) between different threads make use of the underlying OS primitives. Worst case latency can be made as small as the underlying OS permits by carefully distributing OS threads in the system and setting different priorities for them. To be able to use the threaded integration a deployment diagram has to be created, see The Deployment Editor.

For a more in depth discussion of the different integration types, please refer to the Cadvanced documentation on the matter, Different Integration Models, since they are very similar.

Essential files

The essential files used for compilation and linking of an Cextreme Code Generator application are

- `comp.opt`: Defines commands for compiling, linking etc.
- `make.opt` or `makeoptions`: Contains compiler flags and compilation of the kernel.
- `<compiler and integration dependant name>.m`: This is the generated makefile for the application. The name depends on the

choice of compiler and integration type, e.g. `cl_appl_ce.m` for light integration using Microsofts cl compiler.

- `systemname_env.tpm`: This is the generated template makefile for files that are not under full control of the code generator. This file is generated if template environment functions are generated.

comp.opt

The `comp.opt` file can be used to control the complete build process from a custom build script. The `comp.opt` file consists of five important lines (plus lines counted as comments). The syntax for the file is described in "Creating a New Library" on page 3139, where a more detailed description of the complete make process can also be found. Below an overview for Cextreme Code Generator can be found.

- **Line 1:** The syntax for including `make.opt` in the generated makefile for the application.
- **Line 2:** Template for a compilation command.
- **Line 3:** Template for link command.
- **Line 4:** The command to be executed to start the make process. This should be a shell command that the build script will execute after it has finished the code generation.
- **Line 5:** Template for building a library.

systemname.m

When the code generator has finished the process of generating code it will execute the command defined in `compt.opt` at line 4. This usually is some-thing like:

```
make -f systemname.m sctdir=<a directory>
```

This will invoke the make facility with the generated makefile.

makeoptions (make.opt)

In the beginning of the generated makefile is an include statement, including the file `makeoptions` (`make.opt`) can be found. The variable `sctdir`, passed to make at the command line, is used to point out the directory where to find `makeoptions`.

The make program will process the `makeoptions` file, where it finds a number of settings for compiler, linker, and options for the compilation com-mands for the kernel files. In the generated makefile it will then find the com-pilation commands for the generated files and a command to link the object file to an executable.

system name_env.tpm

The template makefile mentioned above, is used to compile the files that are not directly under the control of the code generator. Such a file is generated if a file with template environment functions is generated. This file will have the name `system name_env.tpm` and handles compilation of the file with template environment functions. Targeting Expert lets you specify what template makefile to use. This can either be the generated one (use the file name *), or a user defined file. The contents of the specified template makefile is copied last into the generated makefile.

Adopting a compiler

If you need to adopt e.g. a cross compiler, you should do that by creating a new directory where a `comp.opt` and a `makeoptions` (or `make.opt`) file are placed. The easiest way is usually to copy existing `comp.opt` and `makeoptions` (or `make.opt`) files to a new directory and modify them.

Integration with Compiler and Operating System

This section describes how to set up a compiler and target platform in order to get an application running.

Integration with a new compiler

There are mainly two aspects when a new compiler should be used. Compiler name and switches, and what include files are needed.

Compiler name and switches

Specifying compiler name and suitable compiler switches are part of the build process. These topics are discussed in the previous section – see Compile and Link an Application.

Include files

The other major aspect is the include files needed for the kernel code and for the generated code, but also include files used for user specific code. If no integration is specified a default section including the `.h` file needed by the kernel and the generated code is used. This follows ISO C specifications on system include files. In this case the following is included (compare with the kernel file `extreme_kern.h`):

```
#if defined(USER_CFG_COMPHDEF)
  #include "comphdef.h"
#else
  /* Use default (ISO-C) */
  #include <string.h>
  #include <stdlib.h>
  #include <limits.h>
  #include <stdarg.h>
  #ifdef CFG_ADD_STDIO
    #include <stdio.h>
  #endif

  #if defined(__cplusplus) && defined(_MSC_VER)
    #include <cstddef> /* C++ and Microsoft compiler
*/
  #else
    #include <stddef.h>
  #endif
#endif
```

That is if `USER_CFG_COMPHDEF` is not defined the include files given above is included.

Note: stdio.h is only needed if printing is used.

If the compiler/run-time library with the compiler does not provide the functions defined in `string.h`, the kernel (`extreme_kern.c`) includes implementations of the function that is used from this file. The functions are:

`memset`, `memcpy`, `strlen`, `strcpy`, `strncpy`, and `strcmp`

By inserting defines from the list below to the `extreme_user_cfg.h` file generated by the Targeting Expert, it is possible to use the kernel implementations of these functions:

```
#define USER_CFG_USE_memset
#define USER_CFG_USE_memcpy
#define USER_CFG_USE_strlen
#define USER_CFG_USE_strcpy
#define USER_CFG_USE_strncpy
#define USER_CFG_USE_strcmp
```

To customize the list of included system files (and possibly other `.h` files needed in your application), the macro `USER_CFG_COMPHDEF` should be defined. This can be done in the Targeting Expert.

Another option, used in the predefined kernels, is to include this definition on the command line to the compiler using a -D option or something similar.

In this case the file `comphdef.h` will be included. It is then also necessary to adopt the build process, specifically the list of directories where the compiler looks for include file, so the compiler finds the correct `comphdef.h`. Usually this is performed by -I options to the compiler.

Integration with the run-time system

There are a number of aspects that is important in the integration with the run-time system provided by the underlying hardware and software:

- Clock function
- Memory management for dynamic memory
- Protecting data by disabling/enabling interrupts (non threaded integration)
- Threaded integration with an RTOS.

Just as for the compiler integration you can enable your own integration by including a define of the macro `USER_CFG_RTAPIDEF` in `extreme_user_cfg.h` (or as a compiler option):

```
#define USER_CFG_RTAPIDEF
```

In that case the file `rtapidef.h` will be included in `extreme_kern.h` and `rtapidef.c` will be included in `extreme_kern.c`. Appropriate options must be given to the compiler so it finds the correct `rtapidef.*` files.

If `USER_CFG_RTAPIDEF` is not defined then a standard integration is used. The algorithm used for this selection is:

```
if (threading is used)
  if (Microsoft or Borland compiler is used)
    Use a Win32 threaded integration
  else
    Use a POSIX pthreads integration
  endif
else
  Use a non-threaded integration
endif
```

Note: POSIX pthreads integration and non-threaded integrations will likely work on most UNIX systems/compilers. This is however tested only on supported systems/compilers.

Clock function

To support the SDL concept of timers, a clock function is necessary. The generated code and the kernel assumes that there is a clock function called `xNow` that returns the current time. Time values are represented by values of type `SDL_Time`.

There are two standard implementations of the clock function, one for UNIX like systems and one for Windows. In Windows the standard function `_ftime` is used to read the system clock, while on UNIX like systems the standard function `clock_gettime` is used.

To implement a clock function you should include your own `rtapidef.h` and `rtapidef.c` files according to the details below.

If timers are not used and the clock is not explicitly accessed in SDL or C, there is no need for a clock implementation. Just include the macro definition:

```
#define xInitSystime()
```

in `rtapidef.h`.

If a clock implementation is needed then include the following prototypes in `rtapidef.h`:

```
extern void xInitSystime(void);

extern SDL_Time xNow (void);
```

If no initialization function is needed then the `xInitSystime` function can be replaced by the macro.

```
#define xInitSystime()
```

In the file `rtapidef.c` the implementation of these functions should be provided. The implementations will depend a lot on the support in software and hardware for the underlying architecture.

Memory management

In some cases dynamic memory is needed by a generated application. To support this an `alloc` and a `free` function must be provided. You have three possibilities:

- The first alternative is to use the built-in memory package. This option can be specified in the Targeting Expert. In this case an array of bytes is defined and the memory in this array is used as dynamic memory. The algorithm used for the memory management is implemented in the kernel and is basically a best fit algorithm. The size of the array can, of course, be set by you. It is also possible to specify a minimum block size. In that case only blocks of `size 2^n * min_block_size` will be allocated. This may reduce the risk for memory fragmentation.
- The second alternative is to rely on a `calloc` and a `free` function provided by the underlying layer. This is the default behavior. If calloc is not available a combination of malloc and memset can be used instead, by defining `CFG_NO_CALLOC_AVAILABLE`.
- The third alternative is to implement the memory management yourself. In that case the macro `USER_CFG_USE_USER_MEMFUNC` should be defined. It is assumed that you implement the functions:

```
extern void *xAlloc (unsigned int);
extern void xFree (void **);
```

in the file `rtapidef.c`. The prototypes are present in `extreme_kern.h` so it is not necessary to insert them into `rtapidef.h`.

Note: If the application does not use dynamic memory, there is no need to implement these functions.

The `xAlloc` function should allocate memory of the size given as parameter and return a reference to the memory. It is assumed that the memory is set to 0 by the `xAlloc` function. Example:

```
void *xAlloc (unsigned int Size)  {
  void * Ptr;
  Ptr = (void *)malloc(Size);
  if (Ptr)
    (void)memset(Ptr, 0, Size);
  return Ptr;
}
```

The `xFree` function takes a pointer to a pointer to some memory to be returned to the pool of free memory. The function should free the memory and set the pointer to 0. Example:

```
void xFree (void ** Ptr)
{
  if (*Ptr) {
    free(*Ptr);
    *Ptr = (void *)0;
  }
}
```

The `xAlloc` and `xFree` functions must be thread safe. The functions in the built-in memory package are protected by a semaphore or by turning off and on interrupts. In case two, when using the OS `calloc` and `free` it is assumed that these functions are thread safe.

Disable and enable interrupts

In a non-threaded application where you want to be able to send signals into the system in interrupt routines, some important data structures for signals must be protected from simultaneous access. To achieve this it is necessary to disable interrupts while executing certain operations in the kernel.

To implement disabling and enabling of interrupts, you should in define two macros in `rtapidef.h` with the structure given below.

```
#define XBEGIN_CRITICAL_PATH \
        UserCodeToDisableInterrupts;
#define XEND_CRITICAL_PATH \
```

```
UserCodeToEnableInterrupts;
```

where `UserCodeToDisableInterrupts` and `UserCodeToEnableInterrupts` should be replaced by code performing these actions for the hardware and software platform that is used.

Threaded integrations

To implement a new integration it is recommended to use this manual together with the code for some existing integration(s). There are some major aspects that have to be handled to implement an integration with real-time operating system.

- It is necessary to implement a clock function.
- There is need for a number of mutexes or binary semaphores to protect some shared data.
- Some startup code, for creating threads with relevant properties and synchronizing them are needed.
- A thread must be able to suspend its execution when it is idle. It must then be possible to wake it up again when a signal is sent to a part in the thread.

To explain the details in these integration aspects the POSIX integration will be used as an example. Apart from the code mentioned below the `rtapidef.h` should include the necessary system include files to be able to access the concepts needed.

Example 632: Includes in rtapidef.h for POSIX

```
#include <pthread.h>
#include <sched.h>
#include <semaphore.h>
#include <time.h>
#include <sys/time.h>
```

If the RTOS has any requirements on the `main()` function, which might be the case, it is possible to rename the `main()` function included in `extreme_kern.c` by defining `XMAIN_NAME` to for example:

```
#define XMAIN_NAME cextreme_main
```

Then the user has to implement a proper main function that calls the `cextreme_main` function.

The clock function

The clock function should be implemented according to the description found in a previous section.

Protection of shared data

It is necessary to protect the list of available signals, the list of available timers, the list of free parts (for create actions), and, if the memory package is used, the memory used by the package.

For this four global mutexes or binary semaphores are needed. These variables should be defined extern in `rtapidef.h` and declared in rtapidef.c. The names of the variables should be the same as in the example given below.

Example 633: In rtapidef.h:

```
extern pthread_mutex_t xFreeSignalMutex;
extern pthread_mutex_t xFreeTimerMutex;
extern pthread_mutex_t xCreateMutex;
#ifdef USER_CFG_USE_MEMORY_PACK
  extern pthread_mutex_t xMemoryMutex;
#endif
```

Example 634: In rtapidef.c:

```
pthread_mutex_t xFreeSignalMutex;
pthread_mutex_t xFreeTimerMutex;
pthread_mutex_t xCreateMutex;
#ifdef USER_CFG_USE_MEMORY_PACK
  pthread_mutex_t xMemoryMutex;
#endif
```

These four variables should be initialized during the startup of the application to an unlocked state. The function `xThreadInit` is a proper place for this initialization.

Example 635: xThreadInit

```
void xThreadInit (void) {
  (void)pthread_mutex_init(&xFreeSignalMutex, 0);
  (void)pthread_mutex_init(&xFreeTimerMutex, 0);
  (void)pthread_mutex_init(&xCreateMutex, 0);
  #ifdef USER_CFG_USE_MEMORY_PACK
    (void)pthread_mutex_init(&xMemoryMutex, 0);
  #endif
  ....
}
```

The lock and unlock operation must also be implemented for mutexes or binary semaphores. The following two functions should be implemented.

Example 636: Functions for lock and unlock

In `rtapidef.h`:

```
extern void xThreadLock (pthread_mutex_t *);
extern void xThreadUnlock (pthread_mutex_t *);
```

In `rtapidef.c`:

```
void xThreadLock (pthread_mutex_t *M)
{
  (void)pthread_mutex_lock(M);
}
void xThreadUnlock (pthread_mutex_t *M)
{
  (void)pthread_mutex_unlock(M);
}
```

Startup phase - creating the threads

After some basic initialization the Cextreme Code Generator kernel will start the specified threads in the `main()` function.

For each thread the functions `xThreadInitOneThread` and `xThreadStartThread` will be called, where `xThreadInitOneThread` should perform some thread specific initialization and `xThreadStartThread` should start the thread. Each thread should run the function `xMainLoop` declared in the kernel. This is performed by using a wrapper function, `xThreadEntryFunc`, which is defined in the integration and is the function that is actually started in the thread.

After all the threads have been started the function `xThreadGo` is called in the function `main()`. Some more information on these functions are given below.

It is important that the started threads do not execute any SDL transitions before all threads are created. Therefore the `xThreadEntryFunc` will as first action wait on a semaphore. The `xThreadGo` function will when all threads are created release this semaphore.

The global data structure `xSysD` is an array with components of type `xSystemData`, with one component per thread. `xSysD` contains global information about what is going on just now in the threads. Details about the information in `xSysD` can be found in the section Overview of

Important Data Structures. In the context of RTOS integrations two aspects are important. Each thread (the `xMainLoop` function) must know the address for the component in `xSysD` representing the thread. Each `xSysD` component contains a field of type `xThreadVars`, which should be defined in the RTOS integration.

Example 637: xThreadVars type in rtapidef.h

```
typedef struct {
  pthread_mutex_t SignalQueueMutex;
  pthread_cond_t  SignalQueueCond;
  pthread_t       ThreadId;
} xThreadVars;
```

where the two first fields will be discussed in the next section, and the `ThreadId` will be used during the startup phase to store the identity of the threads.

The code for the behavior described in this section should look something like the following example.

Example 638:

In rtapidef.h:

```
extern sem_t xInitSem;
#if !defined(USER_CFG_USE_xInEnv) && !defined(XENV)
  extern sem_t xMainThreadSem;
#endif
extern void xThreadInitOneThread (
      struct _xSystemData *);
extern void xThreadStartThread (
      struct _xSystemData *,
      unsigned int, unsigned int,
      unsigned int, unsigned int);
```

In rtapidef.c:

```
sem_t xInitSem;
#if !defined(USER_CFG_USE_xInEnv) && !defined(XENV)
  sem_t xMainThreadSem;
#endif

void xThreadInit (void)
{
  ....
  (void)sem_init(&xInitSem, 0, 0);
}

void xThreadInitOneThread(struct _xSystemData
```

```
*xSysDP)
{
  (void)pthread_mutex_init(
      &xSysDP->ThreadVars.SignalQueueMutex, 0);
  (void)pthread_cond_init(
      &xSysDP->ThreadVars.SignalQueueCond, 0);
}

static void *xThreadEntryFunc (void *xSysDP)
{
  (void)sem_wait(&xInitSem);
  (void)sem_post(&xInitSem);
  xMainLoop((xSystemData *)xSysDP);
}

void xThreadStartThread(struct _xSystemData *xSysDP,
      unsigned int StackSize, unsigned int Prio,
      unsigned int User1, unsigned int User2)
{
  pthread_attr_t Attributes;
  ....
  (void)pthread_create(&xSysDP->ThreadVars.ThreadId,
        &Attributes, xThreadEntryFunc, (void
*)xSysDP);
  ....
}

void xThreadGo(void)
{
  (void)sem_post(&xInitSem);
  #if defined(USER_CFG_USE_xInEnv)
    xInEnv();       /* Cextreme  */
  #elif defined(XENV)
    xInEnv(xNow()); /* Cadvanced */
  #else
    (void)sem_init(&xMainThreadSem, 0, 0);
    (void)sem_wait(&xMainThreadSem);
  #endif
}
```

The `xInitSem` semaphore is used for synchronization of the threads. It is initialized in the beginning of `xThreadInit` to 0, i.e. to a blocking state. After that the `xThreadStartThread` once for each thread that is to be started. The function `pthread_create` will call the function given as third parameter (`xThreadEntryFunc`) with the `void *` parameter given as fourth parameter (the `xSysD` pointer) as parameter. `pthread_create` will also store the identity of the thread in the variable passed as first parameter. The second parameter is the properties of the thread. This will be discussed later in this section.

If any of the threads get a chance to execute before all the threads are created, these threads will hang on the `sem_wait` call in `xThreadEntryFunc`, until the main thread calls `xThreadGo` that will post the semaphore `xInitSem` once. One of the threads waiting on this semaphore will then be able to execute and will immediately post the semaphore again. This will continue until all threads are free to execute.

After that all threads are running and depending on the OS and the application properties, the main thread can perform different things. The recommendation is to call the `xInEnv` function and let that function run in this thread. Another alternative is to hang the main thread on a semaphore, as shown above using the semaphore `xMainThreadSem` (if `xInEnv` is not used). In this case you can post the `xMainThreadSem` semaphore anywhere to restart the execution of the main thread.

When the main thread returns from the function `xThreadStart`, the program will continue to execute in the `main()` function and will perform a call to `exit`. The behavior of a threaded program when the main thread performs exit is OS dependent. In POSIX pthreads all threads are stopped at such an action. That is the reason why it is important to hang the main thread at the end of the `xThreadStart` function.

Now to the properties of the threads. In most RTOS, properties like stack size and priority can be set for individual threads.

- The first value is interpreted as the stack size.
- The second value is interpreted as the priority.
- The third and fourth values can be used for other properties, defined by the RTOS integration or defined by you.

The currently predefined integrations only makes use of the first and second values. These values are passed as parameters to the `xTreadStartThread` function.

How the properties are set up in detail depend on the RTOS. Compare with the available integrations, in the function `xThreadStartThread`, for examples.

In `rtapidef.h` proper default values for the four `xThreadData` fields should be set up. These default values are used if no value is specified in the thread definition.

Example 639:

```
#define DEFAULT_STACKSIZE     1024
#define DEFAULT_PRIO             0
#define DEFAULT_USER1            0
#define DEFAULT_USER2            0
```

Suspending and waking up threads

When a thread finds out that it has nothing more to do, at least just for the moment, it should suspend itself to make the processor available for other threads. The thread should then wake up again either when a timer has expired and needs to be handled, or when some other thread (including `xInEnv`) sends a signal that should be treated by the suspended thread.

To implement these features one mutex (binary semaphore) is used together with some sort of conditional variable. You need the possibility to perform a condition wait, with or without a time-out. You also need a way to signal to a thread to wake up again. These two entities are needed for each thread and is therefore included in the `xThreadVars` struct mentioned earlier:

```
typedef struct {
  pthread_mutex_t SignalQueueMutex;
  pthread_cond_t  SignalQueueCond;
  pthread_t       ThreadId;
} xThreadVars;
```

The purpose of the `SignalQueueMutex` is to protect the signal queue where signals from the outside of the thread are inserted (`ExternSignalQueue` in the `xSysD` array). The `SignalQueueCond` should facilitate the conditional wait.

The `SignalQueueMutex` should be initialized in `xThreadInitOneThread`. If `SignalQueueCond` needs to be initialized it could be performed at the same place.

Example 640:

```
void xThreadInitOneThread(struct _xSystemData
*xSysDP)
{
  (void)pthread_mutex_init(
             &xSysDP->ThreadVars.SignalQueueMutex,
             0);
  (void)pthread_cond_init(
             &xSysDP->ThreadVars.SignalQueueCond,
             0);
```

```
}
```

The `SignalQueueMutex` is locked by using the function `xThreadLock`, discussed above. It is then unlocked in three different ways:

- `xThreadUnlock` (discussed above)
- `xThreadWaitUnlock`
- `xThreadSignalUnlock`

The `xThreadWaitUnlock` is called by the thread itself when it has come to the conclusion that it should suspend itself, while `xThread-SignalUnlock` is called by another thread that wants to wake up the current thread. Both functions are passed the `xSysD` pointer for the thread that the operation should be performed on.

Example 641:

In `rtapidef.h`

```
extern void xThreadWaitUnlock (
               struct _xSystemData *);
extern void xThreadSignalUnlock (
               struct _xSystemData *);
```

In `rtapidef.c`:

```
void xThreadWaitUnlock (struct _xSystemData *xSysDP)
{
  #if defined(CFG_USED_TIMER) || defined(THREADED)
    #ifdef THREADED
      /* Cadvanced */
      if (xSysDP->xTimerQueue->Suc ==
          xSysDP->xTimerQueue) {
    #else
      /* Cextreme */
      if (! xSysDP->TimerQueue) {
    #endif
        (void)pthread_cond_wait(
            &xSysDP->ThreadVars.SignalQueueCond,
            &xSysDP->ThreadVars.SignalQueueMutex);
      } else {
        struct timespec timeout;
        #ifdef THREADED
          /* Cadvanced */
          timeout.tv_sec =
            ((xTimerNode)xSysDP->xTimerQueue->Suc)->
          TimerTime.s;
          timeout.tv_nsec =
            ((xTimerNode)xSysDP->xTimerQueue->Suc)->
            TimerTime.ns;
        #else
          /* Cextreme */
```

```
            timeout.tv_sec = xSysDP->TimerQueue->
              Time.s;
            timeout.tv_nsec = xSysDP->TimerQueue->
              Time.ns;
          #endif
          (void)pthread_cond_timedwait(
            &xSysDP->ThreadVars.SignalQueueCond,
            &xSysDP->ThreadVars.SignalQueueMutex,
            &timeout);
        }
    #else
        (void)pthread_cond_wait(
          &xSysDP->ThreadVars.SignalQueueCond,
          &xSysDP->ThreadVars.SignalQueueMutex);
    #endif
    (void)pthread_mutex_unlock(
      &xSysDP->ThreadVars.SignalQueueMutex);
}

void xThreadSignalUnlock (struct _xSystemData
                            *xSysDP)
{
  (void)pthread_cond_signal(
    &xSysDP->ThreadVars.SignalQueueCond);
  (void)pthread_mutex_unlock(
    &xSysDP->ThreadVars.SignalQueueMutex);
}
```

At the time when `xThreadWaitUnlock` or `xThreadSignalUnlock` is called the `SignalQueueMutex` will be locked and must therefore be unlocked at the end of both functions.

In `xThreadWaitUnlock` the thread wants to suspend itself. If timers are used and there is a timer active in the timer queue, it should wait until the timer expires or until some other thread tells it to wake up. In POSIX pthreads the function `pthread_cond_wait` performs exactly this. If timers are not used or there is no timer active, the thread should be suspended until someone else wakes it up. In POSIX pthreads this can be achieved with the function `pthread_cond_wait`.

In `xThreadSignalUnlock` the thread given by the parameter should be waken up. Here the pthread function `pthread_cond_signal` can be used.

The integrations described here are also used when the C Code Generator is used to generate threaded applications. This adds a few requirements in the implementation of a threaded integration. First a function that can stop a thread is needed.

In rtapidef.h:

```
#if defined(THREADED) || \
    defined(CFG_USED_DYNAMIC_THREADS)
  extern void xThreadStopThread(struct _xSystemData
*);
#endif
```

In rtapidef.c:

```
#if defined(THREADED) || \
    defined(CFG_USED_DYNAMIC_THREADS)
void xThreadStopThread(struct _xSystemData *xSysDP)
{
  pthread_mutex_destroy(
      &xSysDP->ThreadVars.SignalQueueMutex);
  pthread_cond_destroy(
      &xSysDP->ThreadVars.SignalQueueCond);
  pthread_exit(0);
}
#endif
```

`THREADED` is defined when using the C Code Generator but not when using the Cextreme Code Generator. The `xThreadStopThread` function should clean up the thread specific semaphores and stop the thread. It is always the thread that should be stopped that will call this function to stop itself.

Another difference is the way timers are accessed for the two code generators. This affects the details in the `xThreadWaitUnlock` function.

In the case of the C Code Generator the RTOS integrations are accessed through a macro layer. The macros in this layer is used in the C Code Generator kernel files and in the generated code.

Example 642: Defines in scttypes.h

The following defines are relevant (from `scttypes.h`):

```
#define THREADED_GLOBAL_VARS
#define THREADED_GLOBAL_INIT \
      xThreadInit();
#define THREADED_THREAD_VARS \
      xThreadVars ThreadVars;
#define THREADED_THREAD_INIT(SYSD) \
      xThreadInitOneThread(SYSD);
#define THREADED_THREAD_BEGINNING(SYSD)
#define THREADED_AFTER_THREAD_START \
      xThreadGo();
#define THREADED_START_THREAD(F, SYSD, \
                              STACKSIZE, PRIO, \
                              USER1, USER2) \
      xThreadStartThread(SYSD, STACKSIZE, \
                         PRIO, USER1, USER2);
#define THREADED_STOP_THREAD(SYSD) \
```

```
      xThreadStopThread(SYSD);
#define THREADED_LOCK_INPUTPORT(SYSD) \
      xThreadLock( \
            &SYSD->ThreadVars.SignalQueueMutex);
#define THREADED_UNLOCK_INPUTPORT(SYSD) \
      xThreadUnlock( \
            &SYSD->ThreadVars.SignalQueueMutex);
#define THREADED_WAIT_AND_UNLOCK_INPUTPORT(SYSD) \
      xThreadWaitUnlock(SYSD);
#define THREADED_SIGNAL_AND_UNLOCK_INPUTPORT(SYSD) \
      xThreadSignalUnlock(SYSD);
#define THREADED_LISTREAD_START \
      xThreadLock(&xFreeSignalMutex);
#define THREADED_LISTWRITE_START \
      xThreadLock(&xFreeSignalMutex);
#define THREADED_LISTACCESS_END \
      xThreadUnlock(&xFreeSignalMutex);
#define THREADED_EXPORT_START \
      xThreadLock(&xCreateMutex);
#define THREADED_EXPORT_END \
      xThreadUnlock(&xCreateMutex);
```

Optimization and Configuration

Optimization and configuration is mainly performed in two `.h` files. These are `auto_cfg.h` and `extreme_user_cfg.h`. The file `auto_cfg.h` is generated by the Cextreme Code Generator, and contains the optimization and configuration that can be automatically computed given the system that code is generated for. The `extreme_user_cfg.h` file is generated by The Targeting Expert from configuration information provided by the user.

auto_cfg.h

The `auto_cfg.h` has the contents discussed below. The first section in the file is used to configure mainly the size of some arrays and the size (8, 16, or 32 bits) needed to represent certain entities.

```
#define CFG_NUMBER_PROCESS_TYPES 2
#define CFG_MAX_INSTANCES 1
#define CFG_NUMBER_TIMERS 0
#define CFG_MAX_TIMER_INSTS 0
#define CFG_NUMBER_SIGNALS 4
#define CFG_NUMBER_THREADS 0
#define CFG_MAX_ACTIONS 2
#define CFG_MAX_STATES 1
#define CFG_MAX_STATE_INDEX_ENTRY 2
```

In the second section information is given about what concepts that are used in the system that code is generated for. For concepts that are used a `#define` is generated, while for concepts not used, a comment, saying the corresponding macro is not defined, is generated. The information in this section is used to scale the code, to remove code and fields in data structures that is not needed.

```
#define CFG_USED_DYNAMIC_THREADS
#define CFG_USED_UNLIMITED_INSTANCES
#define CFG_USED_TIMER
#define CFG_USED_CREATE
#define CFG_USED_STOP
#define CFG_USED_SAVE
#define CFG_USED_STATE_STAR
#define CFG_USED_INPUT_SAVE_STAR
#define CFG_USED_GUARD
#define CFG_USED_SIGNAL_WITH_PARAMS
#define CFG_USED_TIMER_WITH_PARAMS
#define CFG_USED_SIGNAL_WITH_DYN_PARAMS
#define CFG_USED_SENDER
#define CFG_USED_OFFSPRING
#define CFG_USED_PARENT
#define CFG_USED_SELF
```

```
#define CFG_USED_PROCEDURE
#define CFG_USED_RPC
#define CFG_USED_INITFUNC
```

The third section contains information about items in connection with data types, used to remove code in the file `sctpred.c` not needed by the generated application. The section contains a sequence of defines looking like the example below.

```
#define XNOUSE_"at-lot-of-things-about-data-types"
```

In rare cases the `auto_cfg.h` file might contain information that does not fit the application to be built. One such case is if the third section contains information that a certain function is not used, and therefore removed with ifdefs. The function is, however, used in user provided C code, in for example the environment functions. In such cases it is possible to override the information in `auto_cfg.h` by including macro definitions last in `extreme_user_cfg.h`.

Example 643: Assume the `auto_cfg.h` contains

```
#define XNOUSE_LENGTH_CHARSTRING
```

which shows that the length function for character strings are not used. To override this the following code should be included in `uml_cfg.h`:

```
#ifdef XNOUSE_LENGTH_CHARSTRING
  #undef XNOUSE_LENGTH_CHARSTRING
#endif
```

extreme_user_cfg.h

This file is generated by the Targeting Expert and contains information about the options that you have selected.

`extreme_user_cfg.h` is, just as `auto_cfg.h`, a file that should contain defines that are used to build the Cextreme Code Generator application with specified options. The file `extreme_user_cfg.h` will include `auto_cfg.h`.

Some of the features discussed below can be selected in the Targeting Expert and will affect the contents of the `extreme_user_cfg.h` file. The options not included in the UI can be inserted in the file directly, just make sure that your modifications won't be overwritten by the Targeting Expert.

Process properties

- `#define USER_CFG_USE_NUMBER_FREE_INST <Integer>`

  In case there are parts with no maximum number of instances, then memory will be dynamically allocated at startup for the number of initial instances plus the value given here. The value must be > 0. Default value is 5.

Signal properties

- `#define USER_CFG_MAX_SIGNAL_INSTS <Integer>`

  This defines the length of the static signal queue. If dynamic signals are not used (compare with `USER_CFG_NOUSE_DYNAMIC_SIGNALS`), an error occurs if the signal queue is full at the same time as an attempt to send a signal is made.

- `#define USER_CFG_NOUSE_DYNAMIC_SIGNALS`

  Turn off usage of dynamic memory for signals. This feature should normally be turned off in smaller systems, where dynamic memory is not to be used. In that case it is important to set `USER_CFG_MAX_SIGNAL_INSTS` to an appropriate value.

- `#define USER_CFG_SIGNALS_NEVER_DISCARDED`

  Define to remove the code for freeing signal parameters in case a signal is discarded. Only applicable if signals with dynamic parameters are used. If `XMK_USED_SIGNAL_WITH_DYN_PARAMS` is defined in `auto_cfg.h`. If `USER_CFG_SIGNALS_NEVER_DISCARDED` is defined and a signal with dynamic parameters is discarded there will be a memory leak.

- `#define USER_CFG_USE_SIGNAL_PRIORITIES`

  Use priorities on signals. This means that the signal queue is sorted first in priority order and then in arrival order (when same priority).

- `#define USER_CFG_TIMER_PRIO <Integer>`

  The priority assigned to all timer signals. Default is 50.

- `#define USER_CFG_CREATE_PRIO <Integer>`

  The priority assigned to all startup/create signals. Default is 50.

- `#define USER_CFG_DEFAULT_PRIO <Integer>`

The priority assigned to all signals not having an explicit priority. Default is 50.

- `#define USER_CFG_MSG_BORDER_LEN <Integer>`

  This defines the length of the data field for parameters within the signal. If the signal parameters do not fit into this memory dynamic memory allocation is used. Default is 4 if signals with parameters is used (`XMK_USED_SIGNAL_WITH_PARAMS` is defined) and 0 if no signals have parameters.

Timer properties

- `#define USER_CFG_MAX_TIMER_INSTS <Integer>`

  Length of static timer queue. The default length is the computed value `CFG_MAX_TIMER_INSTS` from `auto_cfg.h`, which in case of timers without parameters is the maximum amount of possible active timers. In case of timers with parameters this value can, in extreme cases, be to small. Normally a value smaller than `CFG_MAX_TIMER_INSTS` might be used. If dynamic timers are not used (compare with `USER_CFG_NOUSE_DYNAMIC_TIMERS`), an error occurs if the timer queue is full at the same time as an attempt to set a timer is made.

- `#define USER_CFG_NOUSE_DYNAMIC_TIMERS`

  Turn off usage of dynamic memory for timers. This feature should normally be turned off in smaller systems, where dynamic memory is not to be used. In that case it is important to set `USER_CFG_MAX_TIMER_INSTS` to an appropriate value.

- `#define USER_CFG_TIMER_SCALE <Integer>`

  Scale all time-outs in set actions with this value. This can for example be used during early testing to delay short time-outs to become visible (making for example a 1 millisecond time-out take 1 s). The feature should not be used in a finalized application, as it will cause some speed overhead.

Dynamic memory allocation

- `#define USER_CFG_USE_MEMORY_PACK`

Define to use the memory management package provided in the library, instead of OS functions malloc, free. If this package is used `USER_CFG_MEMORY_SIZE` should also be set (if the default is not appropriate). Default is undefined.

- `#define USER_CFG_USE_MEMORY_PACK <Integer>`

  Define the number of bytes to be used by the memory package. To avoid unnecessary problems define this value as a multiple of 16. Default is 8192 bytes.

- `#define USER_CFG_MEMORY_MIN_BLOCKSIZE <Integer>`

  Define the minimum size of the memory block. If defined only blocks of size:

  2^N * USER_CFG_MEMORY_MIN_BLOCKSIZE, N>=0, are used. This value should be a multiple of 16. Default is undefined.

Error detection

- `#define USER_CFG_ERR_CHECK_BASIC`
  - Turn on checking of basic state machine properties:
  - No more memory for signal sending or create
  - No more memory to allocate the parameters of a signal
  - No more memory for timer instance
  - `xOutEnv()` is not present and signal is sent to the environment
  - Signal is discarded
  - Cannot create more instances as maximum limit reached
  - Public attribute access error

  Default is undefined.

- `#define USER_CFG_ERR_CHECK_INDEX`

  Test that array index is in range. Default is undefined.

- `#define USER_CFG_ERR_CHECK_RANGE`

  Test that syntype values are in range. Default is undefined.

- `#define USER_CFG_ERR_CHECK_PREDEF_O`

  Test error situations in predefined operators. Default is undefined.

- `#define USER_CFG_ERR_CHECK_DECISION`

  Test that there is a path for the current decision value. Default is undefined.

- `#define USER_CFG_ERR_CHECK_NULL_PTR`

  Test that pointers are not NULL before de-referencing. Default is undefined.

- `#define USER_CFG_ERR_CHECK_MEMORY_PACK`

  Test for error situations in the memory package. Default is undefined.

- `#define USER_CFG_ERROR_MESS_STDOUT`

  Define if error messages should be printed on sdtout. Default is undefined.

- `#define USER_CFG_ERROR_MESS_STDERR`

  Define if error messages should be printed on stderr. Default is undefined.

- `#define USER_CFG_USE_ERR_MESS`

  Define if error messages, not only error numbers, should be printed. Default is undefined.

- `#define USER_CFG_WARN_ACTION`

  If defined the user provided function:

  ```
  void xUserWarnAction (unsigned char WarningNumber);
  ```

  is called in case of a warning. Default is undefined.

- `#define USER_CFG_ERR_ACTION`

  If defined the user provided function:

  ```
  void xUserErrAction (unsigned char WarningNumber);
  ```

  is called in case of an error. Default is undefined.

Miscellaneous

- `#define USER_CFG_USE_xInitEnv`
- `#define USER_CFG_USE_xCloseEnv`
- `#define USER_CFG_USE_xInEnv`
- `#define USER_CFG_USE_xOutEnv`

  Define the above to include calls of the corresponding environment function. Default is undefined.

- `#define USER_CFG_ADD_STDIO`

  Define if `stdio.h` should be included. This is automatically performed if any of `USER_CFG_UML_TRACE_STDOUT`, `USER_CFG_ERROR_MESS_STDOUT`, `USER_CFG_ERROR_MESS_STDERR` is defined. Default is undefined.

- `#define USER_CFG_UML_TRACE_STDOUT`

  Define if trace on SDL level should be printed on stdout. Default is undefined.

- `#define XUSE_TYPEINFONODE_CFG`

  Enables the removal of unused type info nodes. Type info nodes are data structures that are used during run-time by the functions providing generic implementations of SDL operators. By removing unnecessary type info nodes the memory footprint of the system can be greatly reduced.

Code generation commands

There are some options in the Targeting Expert that is passed to the code generator, sdtsan, and does not affect the `extreme_user_cfg.h` file. The purpose is to be able to generate as readable code as possible, which is also the default. If certain features are needed, that will affect the readability, these features can be selected individually. These options can be found in the “SDL to C compiler”-tab, in the group “Code generation options”.

- Amount of comments (`Set-Extreme-C-Comments`)

In the code generated for state machines it is possible to select the amount of comments added to the code. The available choices are:

- **1. Off**: Only some comments used to identify transitions are included.
- **2. Structure**: As 1 plus a comment for each translated SDL symbol.
- **3. Full**: As 2 plus comments giving some explanations to the code.

- Generate Run Time Tests (`Set-Extreme-C-Run-Time-Test`)

  This feature must be selected if any of the run-time tests `USER_CFG_ERR_CHECK_INDEX`, `USER_CFG_ERR_CHECK_RANGE`, `USER_CFG_ERR_CHECK_DECISION`, or `USER_CFG_ERR_CHECK_NULL_PTR` is selected.

- Generate Sdt references (`Set-Extreme-C-Sdt-Ref`)

  Include references to SDL source in generated code as C comments. Such references can be used to navigate back to the SDL source.

- Generate trace code (`Set-Extreme-C-Trace`)

  Selecting this feature will tell the code generator to include calls to trace functions (`xTrace` functions) in the generated code. This must be enabled if `USER_CFG_UML_TRACE_STDOUT` is enabled.

- Use Signal Priorities (`Set-Extreme-C-Signal-Prio`)

  This will generate signals with priorities. The priority is an integer value where a lower number means a higher priority.

Overview of Important Data Structures

All the important types discussed in this section are defined in `extreme_kern.h`. Please have that file ready at hand while reading this section.

In generated code there is information about each part in the system. For each part a struct variable of type `xPartTable` is generated. Example:

```
xPartTable xPartData_p_01 = {
  (xInstanceData *)xInstData_p_01,
  sizeof(yVDef_p_01),
  1,
  1,
  (xIniFunc)yIni_p_01,
  (xTransFunc)yPAD_p_01,
  xTransitionData_p_01,
  xStateIndexData_p_01
#ifdef CFG_USED_GUARD
  , 0
#endif
};
```

There is also in generated code an array containing the addresses to the `xPartTable` structs for all parts in the system. This global variable is used at many places to access information about parts. Example:

```
xPartTable *xPartData[] = {
  &xPartData_p_01,
  &xPartData_q_02
};
```

Looking through the contents of the xPartTable struct, it contains the following components:

Component	Description
`InstanceData`	A pointer to an array with one element for each instance of the part. These components are used to store instance specific values of different kinds, like state and local variable values.
`DataLength`	The size of each array element mentioned for the previous component.
`MaxInstances`	The maximum number of concurrent instances of this part.
`InitialInstances`	The number of instances at system start up.

Component	Description
yIni_Function	A reference to a function that initializes the at–tributes of the active class.
yPAD_Function	A reference to the function that implements the state machine for the part.
TransitionTable and StateIndexTable	Tables (arrays) used to determine how to handle a certain signal in a certain state.
GuardFunc	A reference to a function that can compute guard expressions used in the state machine.
ThreadNumber	The thread that this part belongs to. Only used in threaded integrations.

Other important global data structures are the array of available signals and timers. These are

```
/* Signal array */
static xSignal
    xSignalArray[CFG_STATIC_SIGNAL_INSTS];

/* Pointer to first element in list
   of free signals */
static xSignal *xFreeSignalList;

/* Timer array */
static xTimer
    xTimerArray[CFG_STATIC_TIMER_INSTS];

/* Pointer to first element in list
   of free timers */
static xTimer *xFreeTimerList;
```

These variables can be found in `extreme_kern.c` and define one array of signals and array of timers, together with starting pointers for the lists of available signals and timers.

The last important global data structure is the `xSystemData` variable called `xSysD`. In a non-threaded application `xSysD` is a variable of type `xSystemData`, while in a threaded application `xSysD` is an array with components of type `xSystemData`, with one component per thread.

xSysD contains global information about what is going on just now in the application or thread. The following components can be found in xSysD:

Component	Description
CurrentPid	This is the Pid value for the currently executing instance in the application or thread.
CurrentSymbol-Nr	This is the symbol number where the execution should start in the function implementing the state machine. The selection is performed by a switch in the beginning of the function.
CurrentData	This is a pointer to the local variables for the executing instance. This reference goes to the appropriate element in the InstanceData component in the xPartTable for the part.
CurrentSignal	This is a pointer to the signal that caused the currently execution transition.
SignalQueue	This is a reference to the first signal in the global signal queue (global for application or thread). The signals in this queue are linked together in a linked list.
ExternSignal-Queue	This is a signal queue where signals coming from other threads or from the environment first are stored. At well defined points (between transitions) signals are moved to the SignalQueue.
OutputSignal	Variable used as temporary variable while sending a signal in generated code.
TimerQueue	This is a reference to the first timer in the global timer queue (global for application or thread).
ThreadVars	In a threaded application this is a struct containing data about the thread itself. The contents depend on the integration.

Some other data structures that are worth discussing in more detail are Pid values, and contents of signals and timers.

A Pid value is a reference to an executing instance. The Pid value consists of two parts, the part number (parts are numbered 0, 1..., which is used to index the global `xPartData` array) and the instance number (which is used to index the `InstanceData` array in the `xPartTable` for the part). The Pid type is an unsigned type of suitable size to contain these two values.

A signal is defined by the type `xSignal` and contains the following components.

Component	Description
`Next`	Pointer used to link signals in lists.
`Sid`	The identity of the signal (signal type)
`Receiver`	The Pid value of the receiver.
`Sender`	The Pid value of the sender.
`SaveState`	Used to speed up handling of saved signals.
`Prio`	The signal priority, if you have set up to use signal priorities.
`ParPtr`	A pointer to the parameters of the signal. Refers either to ParArea or to allocated memory.
`ParArea`	This is the inline area for signal parameters used if the parameters fit into this area.

A timer is defined by the type `xTimer` and contains the following components:

Component	Description
`Next`	Pointer used to link timers in lists.
`Sid`	The identity of the timer (timer type)
`Owner`	Pid value for the owner of the timer
`Time`	The time set for the timer.
`TimerParValue`	Optional integer parameter for timers.

Chapter

69 *SOMT Introduction*

This chapter describes the techniques that the SOMT method is based on. It also gives a first brief description of the method and the activities it consists of.

Last in the chapter is a list of references, which are referred to throughout this volume.

Background

Object-oriented analysis is a well-known and popular technique for understanding a problem and analyzing a system. There are many different versions of object-oriented analysis published as different methods in various books. The Object Management Group (OMG) has agreed upon a standard language for object-oriented methods and tools - the *Unified Modeling Language* (UML) and its notation ([37], [38]).

SDL [24] and MSC [26], on the other hand, are standardized notations used for specification and design of systems. They have a solid foundation in terms of support for design, formal verification and code generation. Especially for distributed, reactive, real-time applications SDL and MSC has a successful track record.

The scope of this volume is to present a method that integrates object-oriented analysis and SDL design with a special focus on application areas where SDL is known to be suitable. The intention is of course to provide developers with the best of both worlds: The strength of the object oriented analysis in the early phases together with the strong back-end provided by SDL and SDL tools.

The method has been given the name SOMT (short for *SDL-oriented Object Modeling Technique).* As the name indicates the method is essentially an adaptation of OMT to the requirements given by the special application area of distributed, reactive, real-time systems together with the usage of SDL for the design.

Another object-oriented analysis method that has provided input to SOMT is the Objectory method by Jacobsson [19]. In particular, the idea of use cases that originated from this method has been incorporated into SOMT.

A special focus of SOMT is to provide solutions that can be directly applied in a development project. For instance, it provides guidelines about how to apply OMT in a way that enables a smooth transition between analysis and design.

The volume is focused on how to use the different notations and concepts of object-oriented analysis and SDL and does not provide a detailed description on the specific notations. It is therefore recommended that you are at least reasonably familiar with both the general concepts of object-oriented analysis, the OMT notation and SDL before reading this report. [21] is a good introduction to the OMT notation and there exist several text books on SDL, for example [27].

The rest of this document is structured as follows:

- This chapter, *SOMT Introduction*, gives an overview of the SOMT method and introduces the different activities
- chapter 70, *SOMT Concepts and Notations* describes briefly the different concepts and notations that SOMT uses
- chapter 71, *Requirements Analysis* covers the requirements analysis activity
- chapter 72, *System Analysis* presents the system analysis activity
- chapter 73, *From Analysis to Design* discusses the steps and considerations when moving from analysis to design
- chapter 74, *System Design* moves on to system design
- chapter 75, *Object Design* deals with object design
- chapter 76, *SOMT Implementation* ends the SOMT activities with a description of the implementation activity
- chapter 77, *SOMT Projects* focuses on how to organize a SOMT project

Overview of the SOMT Method

The SOMT method provides a framework that shows how to use object-oriented analysis and SDL-based design together in a coherent way. The framework is based on describing the analysis and design of a system as a number of activities. Each activity deals with some specific aspects of the development process. The work done in the activities is centered around a number of models, that document the result of the activities. As an integral part of the SOMT method, guidelines are given for the transition between the different models. The activities and the main models used in SOMT are illustrated in Figure 614.

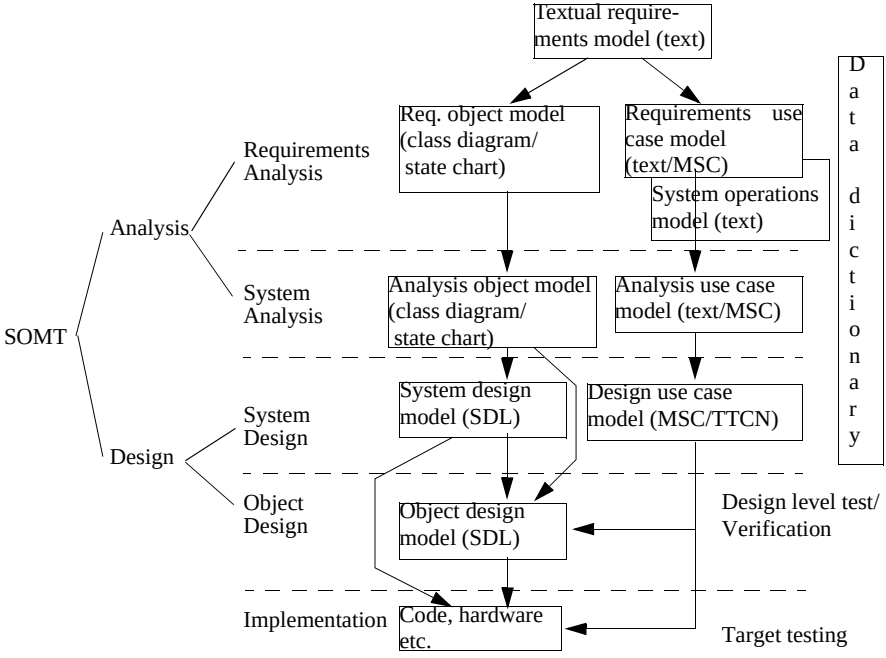


Figure 614: SOMT activities and main models

As can be seen in Figure 614, SOMT consists of five major activities:

- *Requirements analysis.* The purpose of the requirements analysis is to analyze the problem domain and the requirements on the system to be built, essentially by analyzing the system as a black-box in its intended environment.
- *System analysis.* The purpose of the system analysis is to identify the architecture and most important objects that must be represented in the system to achieve the required functionality.
- *System design.* The purpose of the system design phase is to precisely define the architecture of the system including the detailed interfaces between different parts. In the system design the architecture of the system is also analyzed in terms of implementation strategies and decomposition into work packages for different development teams
- *Object design.* The purpose of the object design is to define in detail the functionality including the behavior of all objects.
- *Implementation.* The purpose of this phase is to create the executing application that implements the requirements.

As seen above, SOMT uses a number of different models to describe different aspects of the system. In many cases there are relations between objects in the different models. One especially important relation is that in some cases one object can be seen as an implementation of another object. In SOMT this type of relation is called *implink* (short for implementation link) and corresponds to a design decision taken during the development. The creation and maintenance of implinks forms an important part of the SOMT method. Two points can be made:

- Since the transition between different models is a creative process involving a number of engineering decisions, this is a manual, but tool supported, activity.
- The major tool support is a *Paste As* functionality that will allow you to copy an object in one model and paste it into another model while automatically creating an implink. For example: an object model class from the analysis object model may be pasted as an SDL process type in the system design model.

Note that the *Paste As* functionality can be seen as an implementation of transformation rules between the different models, e.g. from object models to SDL design models.

Another important aspect of implinks is that they facilitate consistency checks between the different models.

It is important to note that even though the description in SOMT of the different activities is a sequential description, this does not mean that in practice these activities can or should be carried out in sequential order. On the contrary, depending on the size and complexity of the application, the different activities can be organized in a number of different ways. This is the topic of the chapter 77, *SOMT Projects*.

Scope of the SOMT Method

The primary focus of the SOMT method is on system development using object oriented analysis and SDL design, with extra support for interface definitions in other notations like ASN.1 [23]. Nevertheless, an important issue in SOMT is the possibility to use the analysis part of SOMT together with design using other notations. Some examples:

- The system to be built is composed of several different parts, where some parts are not suitable to design using SDL. A simple example might be where the system includes a window based user interface that is easier to design using a special purpose tool.
- Another example is when the system includes a data base component. The analysis of the data is suitable to perform using object oriented analysis, but the data base component is preferably designed and implemented using a commercial data base tool.
- Different parts of the system may be designed in different parts of the world and for some reason there has been a choice to use SDL-based development for some parts and other methods for other parts.

The scope of the support given by SOMT when using a design notation other than SDL is, as can be seen in Figure 615, basically:

- The requirements and system analysis of the system are still valid.
- Parts of the system design activity are still supported by SOMT.
- The SDL-based design of components is of course supported.

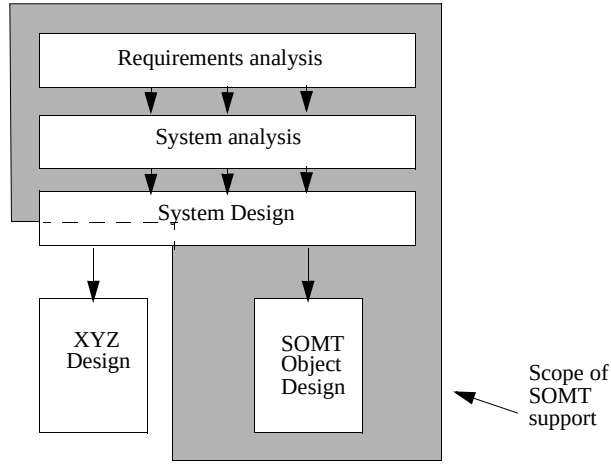


Figure 615: Using SOMT analysis and design together with design using other methods/notations

How much of the system design part that is supported by SOMT is of course very much depending on the “XYZ” method used (see Figure 615). However, the support given by SOMT when using “external” components is at least the definition of the interface of the “external” components. This includes definitions of both static and dynamic aspects as will be seen in chapter 74, *System Design*.

Requirements Analysis

The requirements analysis is the first activity in the SOMT method. The purpose of the activity is, as stated above, to capture and analyze the problem domain and the user requirements on the system to be built. For this purpose the system is viewed as a black-box and only the objects and concepts visible on the system boundary and outside the system are modeled.

There are five major models used in this phase:

- A textual requirements model
- A use case model
- A requirements object model
- A data dictionary

- A system operation model

The textual requirement model is a conventional textual requirements specification. This may come from the customers as input to the development project or it may be created as part of the requirements analysis as a complement to the other models.

The use case model consists of a set of use cases, each described using structured text and/or using MSCs and HMSCs. The purpose of this model is to capture and validate requirements from a user's point of view to make sure that the system will solve the right problem.

The requirements object model is a conventional object model, i.e. one or more diagrams illustrating a number of objects and their relations (including inheritance and aggregation relations). State charts may also be included in the object model to provide high level descriptions of the behavior of important objects. The purpose of the model is two-fold:

- To use context diagrams to describe the system and the external actors that interact with the system
- To document all the concepts found during the requirements analysis and the relations between them in order to ensure that developers and users have a common understanding of the problem domain

The system operation model is a description of the atomic transactions the system must be able to perform. The system operations can be viewed as more detailed specifications of the events that are part of the use cases.

During the requirements phase a data dictionary is also created. The data dictionary is a list of the concepts identified in this phase together with brief explanations of the concepts. It is used and refined throughout all the development phases.

System Analysis

In the system analysis phase the system to be built is analyzed using an object-oriented method. The purpose of the phase is to describe the architecture of the system and identify the objects that are needed to implement the required functionality. The models used in this phase are:

- An analysis object model, to describe the architecture of the system and represent the objects in the system

- An analysis use case model where the interaction between some of the objects is described
- Textual analysis documents to document decisions and architecture related requirements not suitable to be expressed in the object models and use cases

The analysis object model is a conventional object model, consisting of class diagrams and possibly state charts, that forms the input to the object design phase. The analysts should in this model be concerned with identifying the objects that must be present in the system. These objects may come from the requirements object model produced in the requirements analysis, or they may be added in this phase for different reasons. For example, objects may be needed to represent the control logic described by the use cases from the requirements analysis. The differences between the object model in the system analysis phase and the object model in the requirements analysis phase are:

- The requirements analysis object model consists of objects visible on the border of the system and outside the system, e.g. users of the system, while the system analysis object model is focused on the internal object structure of the system.
- The purpose of the object model in the requirements analysis is different from the purpose in the system analysis. The purpose in the requirements analysis is to describe the problem that the system is to solve, while the purpose in the system analysis is to describe the system itself.

System Design

The purpose of the system design is to define the implementation structure of the system and to identify overall design strategies. The models used in this activity are mainly:

- A design module structure, describing the source modules of the design

- An architecture definition, containing
  - SDL system/block diagrams that define the architecture of the resulting application
  - SDL interface definitions of the components of the system using signals or remote procedures
- A design use case model, described by MSC and HMSC diagrams, to define the dynamic interfaces between the different components in the system
- Textual design specifications, describing aspects of the components not suitable for formalization like user interface definitions and other non-functional requirements

The designer of the system will have several different aspects to cover within the system design activity:

- The decomposition of the system into work items, maybe to be implemented by different development teams
- Reuse aspects of the design

Object Design

The purpose of the object design is to create a complete and formally verified description of the behavior of the system. The models used in this phase are essentially SDL diagrams, in particular SDL process graphs that are used to define the behavior of active objects.

One very important aspect of this phase is usage of the *Paste As* concept that will take a user from object models to SDL diagrams. Essentially the designer of the system will in this phase have to consider two major aspects when creating the design model:

- The representation of the analysis object model concepts in the design

  This involves issues like to decide for each object if it is an active or passive object.

- The dynamic behavior of the design objects

The object design also includes a testing/verification task. The purpose of this is to verify that the requirements on the system, expressed e.g. in the design use cases from system design, are correctly implemented in the design.

Implementation

The implementation and testing activities are aimed at producing the final application, i.e. executable software and hardware. The activities in this phase are very much depending on the execution environment of the application but may include:

- Using an automatic code generation tool to produce the code from the SDL design
- Adapting the generated code (the running SDL system) to its environment i.e. regarding signal handling
- Integrating the code to the hardware requirements by means of using real-time operating systems and cross-compilers to generate the executable for the hardware
- Implementing and executing test cases in the target environment based on the design use cases from the system design activity

Summary

SOMT is a method specially designed to suit the development of distributed, reactive, real-time systems. It is based on well-established and industrial-strength techniques:

- object models for analysis
- SDL for design

SOMT recognizes not only a number of models to be made, but also the importance of having a glue between these models. Therefore, SOMT introduces the *Paste As* concept to provide a smooth and well-defined transition between the different models, e.g. between the analysis and design, enabling traceability using *implinks.*

The benefit for a development project is that the analysts and developers all can use the notations best suited for each phase of a development project. The object oriented analysis strength when analyzing requirements and creating conceptual analysis models is combined with the strong back-end given by SDL tools for design, verification and code generation.

References

[18] I. Jacobson, S. Bylund, P. Jonsson, S. Ehnebom, Using contracts and use cases to build plugable architectures, JOOP, May 1995.

[19] I. Jacobson et al, “Object-Oriented Software Engineering”, Addison-Wesley, 1992.

[20] J. Rumbaugh, Getting Started, using use cases to capture requirements, JOOP, September 1994.

[21] J. Rumbaugh et al, “Object-Oriented Modeling and Design”, Prentice Hall, Englewood Cliffs, NJ, 1991.

[22] J. Rumbaugh, OMT: The Development Process, JOOP, May 1995.

[23] ITU Recommendation X.680-683, Abstract Syntax Notation One (ASN.1), 1994.

[24] ITU Recommendation Z.100, Specification and Description Language (SDL), 1994.

[25] ITU Recommendation Z.105, SDL Combined with ASN.1 (SDL/ASN.1), 1995.

[26] ITU Recommendation Z.120, Message Sequence Charts (MSC).

[27] J. Ellsberger, D. Hogrefe, A. Sarma, “SDL – Formal Object-oriented Language for Communicating Systems”, Prentice Hall Europe, 1997, ISBN 0-13-632886-5.

[28] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, J.R.W. Smith, “Systems Engineering using SDL-92”, North-Holland, 1994, ISBN 0-444-89872-7.

[29] F. Belina, D. Hogrefe, A. Sarma, “SDL with Applications from Protocol Specification”, Carl Hansiger Verlag and Prentice Hall International, 1991, ISBN 0-13-785890-6.

[30] D. Coleman et al, “Object-oriented Development - The Fusion Method”, Prentice-Hall, 1994.

[31] OMG, Common Object Request Broker: Architecture and Specification, Revision 2.0, Object Management Group, July 1995.

[32] IBM Rational: IBM Rational SDL and TTCN Suite 6.3 User´s Manual.

[33] G. Booch, “Object Solutions - Managing the Object Oriented Project”, Addison-Wesley, 1996.

[34] ISO Tree and Tabular Combined Notation ISO/IEC 9646-3 1992

[35] Unified Modeling Language, UML Notation Guide version 1.0, (document ad/97-01-09), Object Management Group, January 1997.

[36] David Harel, “Statecharts: a visual formalism for complex systems”, Science of Computer programming (8), 1987.

[37] Unified Modeling Language, UML Notation Guide version 1.1 (document ad/97-08-05), Object Management Group, September 1997.

[38] Unified Modeling Language, UML Semantics version 1.1 (document ad/97-08-04), Object Management Group, September 1997.

Chapter 69 SOMT Introduction

Chapter

70

SOMT Concepts and Notations

This chapter describes the major concepts and notations used in the SOMT method. The notations treated includes object models, state charts, SDL diagrams and Message Sequence Charts (MSCs). Note that the descriptions in this section only briefly describes the different notations. For a more thorough treatment, please consult a textbook on the specific notation.

Activities, Models and Modules

As seen in the previous chapter, the SOMT method describes object oriented analysis and design as a number of activities that produce one or more models. A *model* is here used in an abstract fashion to denote a collection of diagrams, text documents or whatever is needed in the particular model. The concept of a model is used to be able to discuss the results of an activity without going into detail of how the actual diagrams/documents are organized.

A *module*, on the other hand, is the concept used in SOMT to define how the diagrams/documents are organized. The major purpose of the modules is that they form units that should be fairly self-contained and that can be developed by themselves, maybe by different teams. A module is a container of e.g. diagrams and textual documents that has no semantics by itself but that forms a scope unit for names (if this is not given by the notation used). For example, if the Analysis Object Model is described in two modules that both contain a class called “Person”, then these definitions do not refer to the same class. There will be two different “Person” classes, one in each module.

In practice it is beneficial to have a simple mapping between models and modules, either a one-to-one or, if a model is too large, a one-to many mapping where a model is decomposed into several modules.

The actual documentation produced according to SOMT is thus a collection of modules containing diagrams/documents that together form this particular projects representation of the SOMT models.

Implinks and the Paste As Concept

The SOMT method introduces a number of different models that are used to describe different aspects of the system. In particular there are three levels formed by:

- The requirements that describe the problem and external requirements on a system
- The system analysis describing the concepts used in the system
- The system and object design that defines the structure and behavior of the system

One very common relation between objects in different models is that one object can be seen as an implementation of an object in another model. For example, an object in the object model created in the system analysis phase may be reintroduced in the design model as a process type or an ADT (abstract data type). Another example may be an object that was identified in the requirements analysis as something visible on the system boundary, and that later is reintroduced in the analysis object model and finally ends up as a signal in the SDL design. To represent this type of relations among objects the concept of implementation links (*implinks*) is used. An implink is a directed relation between two objects, usually (but not necessarily) in different models. Conceptually we get a picture as in Figure 616.

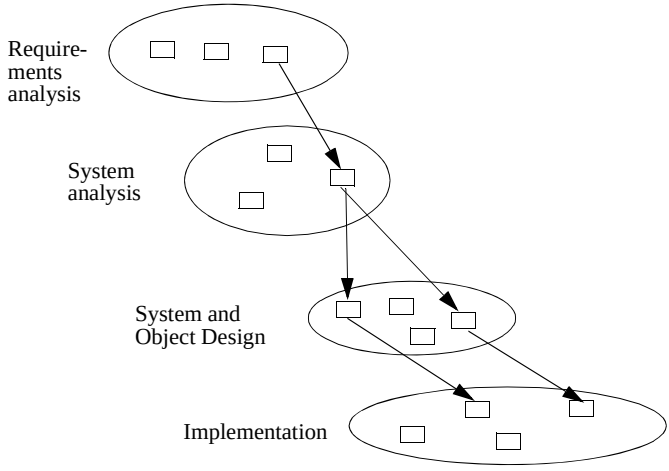


Figure 616: Implinks between objects in different models

If used carefully, the implinks give a possibility to trace requirements all the way down into code. There are several situations where the implinks are very useful:

- *What if* analysis. What are the consequences if a particular requirement is changed?
- Consistency checking. Are all concepts in e.g. the system analysis model implemented?

- Understanding design by following the links backwards from implementation to requirements. What is the purpose of a particular design object? What happens if we change it?

It is important to see that the act of creating an implink is a creative design action that encapsulates a design decision. The *Paste As* mechanism is a special concept used in SOMT to support the task of creating implinks. The idea is that an object in one model can be copied and then *pasted as* a new object in another model, see Figure 617. This action serves both to create the new object and to document the design step using an implink.

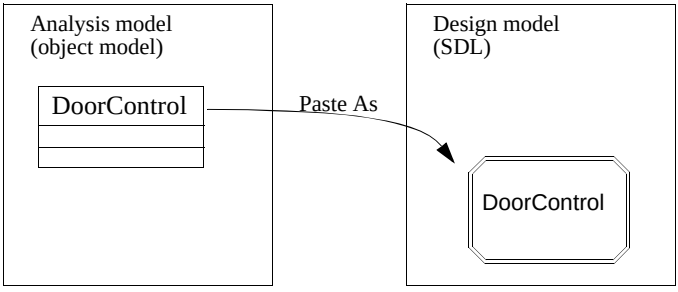


Figure 617: Using Paste As to capture a design step from the system analysis model to SDL

Consistency Checking

The SOMT method uses a number of models and notations and the checking of various aspects of the models is an important part of a development project. This checking can be formulated as a question of identifying “entities” or “concepts” in different models (or in one model), to identify some rules of how these entities should relate to each other, and finally to check that the models are consistent with respect to these rules. This type of checking is in this document called *consistency checking*. Three different types of consistency checking can be identified:

- Checking the internal consistency within one model, e.g. checking that an SDL system is correct with respect to the syntactical rules for SDL

- Checking that two models are consistent with respect to each other and some consistency requirement, e.g. checking that all classes in an object model are described in the data dictionary
- Checking traceability aspects, i.e. checking that the entities in two models are correctly linked (using implinks) to each other. For example, checking that all objects in an analysis model are implemented in the design model

In some sense the checking of traceability aspects is a special case of the checking of the consistency between two models, but it is an important special case and it is given a special treatment in SOMT.

One general observation that can be made is that the identification of concepts/entities is very much depending on the particular notation used. Each separate notation will have to be treated separately following the particular rules that apply to this language. For formal languages like SDL, the concepts and procedures how to find the entities are well defined, while for other languages the rules are different, and for plain, informal text the entity identification will have to be explicitly done by the user. In SOMT there is a special possibility to “mark” words or phrases in text documents. The intention is that this marking means “this concept is important” or “this is a concept that I would like to for consistency checks”.

In the rest of this volume, each activity of the SOMT method together with its associated models will be described in different chapters. Each of these chapters will also include a discussion on consistency rules that are relevant for this particular model.

Object Model Notation

The object model notation from Object Modeling Technique (OMT) and Unified Modeling Language (UML) is a commonly accepted graphical notation used for drawing diagrams that describe objects and the relations between them. The notation that is used in examples in this volume is shown in Figure 618 through Figure 623. For more details about other, more advanced OMT object model concepts, please consult [21], and for details about the UML notation see [37] and [38].

Class

The most important concept in an object model is the class definition. A *class* is a description of a group of similar objects that share the properties defined by the class. The object model notation for a class is exemplified in Figure 618, where the second class definition also shows how to define attributes and operations.

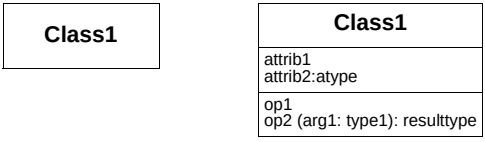


Figure 618: A collapsed class symbol and a class symbol with attributes and operations

In some cases it is necessary to reference classes from an external module. The notation used for this purpose is *ExternalModule::Class*.

Classes may *inherit* attributes and operations from other classes. The object model notation for this is shown in Figure 619.

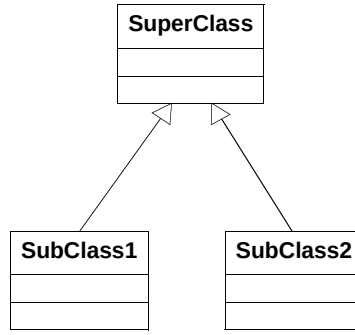


Figure 619: Inheritance between classes

Relations and Multiplicity

Classes may be physically or logically related to each other. This is shown in the object model by means of *associations* as shown in Figure 620. An association may have a name and/or the endpoints of the association may be labeled by the role of this endpoint.

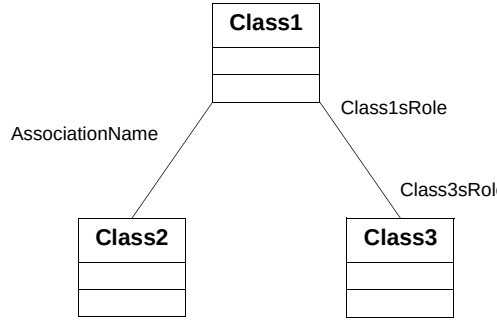


Figure 620: Associations between classes

Aggregation is special kind of association that has its own notation as shown in Figure 621.

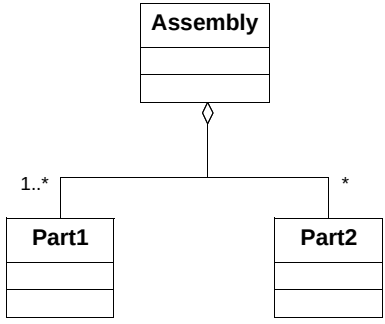


Figure 621: Aggregation

The endpoints of associations and aggregations may have a multiplicity as shown in Figure 622.

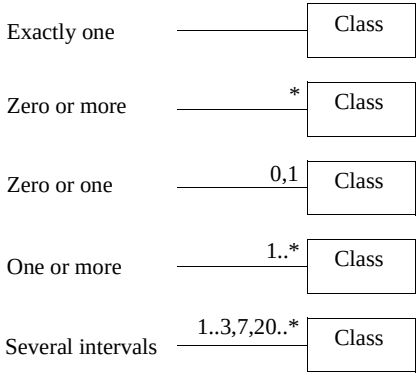


Figure 622: Multiplicity of associations and aggregations

Module

In practise the complete requirements object model is often too big to fit into one diagram. To solve this problem it is possible to use multiple object model diagrams that can be organized into a module, which simply is a list of diagrams. It is important to notice that a class may be present in more than one diagram and still only represent one logical class.

Objects

Besides class definitions, object models may of course also contain objects and their relations. The relation that exists between objects are links, which corresponds to the associations for classes. The object symbol has one field containing the name of the object together with a reference to the class, and an attribute field where constant or default values can be assigned to the object attributes. See Figure 623.

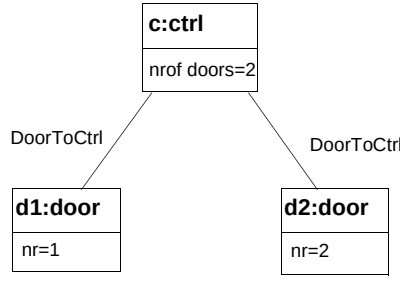


Figure 623: Objects related by links

State Chart Notation

The notation presented in this section is a subset of the notation for state charts presented by David Harel [36] which is used in the Unified Modeling Language (UML) [37] as well as in the Object Modeling Technique (OMT) [21].

A state chart model is suitable to use together with class and object models. The descriptions of the behavior of a class in a class diagram is collected into a state chart which describes the dynamic view of the model by means of states and transitions. The notation to use is presented below.

Notation

State

To describe state charts a state symbol is needed. The state symbol is divided into three compartments, State Name, State Variable and Internal Activity, which are all optional. The top compartment contains the optional name of the state. State symbols with the same name within the same context are considered to be the same. It is not necessary to name states and if several anonymous states exists; each anonymous state symbol is considered to be an individual state. Figure 624 shows a collapsed state and a state with events.

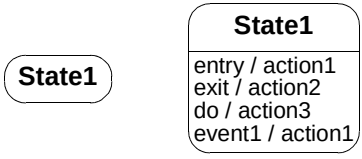


Figure 624: A collapsed state symbol and a state symbol with events

A state symbol may contain an Internal Activity compartment which describes the activities of the current state, activities that are done upon entering the state, activities taking place while in the state and activities executed when exiting the state. Each activity is described in the format:

event-name argument-list '/' action expression

Each event name may appear only once within a state symbol. The event names "entry", "exit" and "do" are reserved and they describe the following actions:

'entry' '/' action expression
An atomic action performed upon entering the state

'exit' '/' action expression
An atomic action performed upon exiting the state

'do' '/' action expression
An action performed during the state

Transition

The second necessary symbol for drawing state charts is the transition. The transition symbol is an arrow which connects two symbols of either type of state, start and termination, see Figure 625.

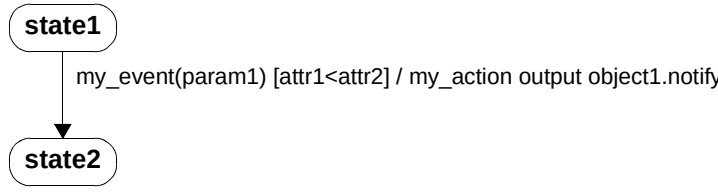


Figure 625: The transition from state1 to state2 is triggered by the event my_event and the condition that attr1 is less than attr2

The syntax for the transition symbol follows the format:

event-signature '[' guard-condition']' '/' action-expression 'output' send-clause

The event-signature consists of the parts:

event-name '(' parameter ',' ... ')'

The guard-condition is a Boolean expression formed by the parameters of the triggering event together with possible attributes and links of the object described by the state chart.

The action-expression describes the action that is executed during the transition. The action may be described by procedures, affected attributes and links.

The send-clause has the format:

destination-expression '.' Destination-event-name '(' argument '.' ...')'

The destination-expression identifies the receiving object or a set of receiving objects.

The Destination-event-name is the name of an event that may be received by the receiving object(s).

Start and Termination Symbol

The start symbol denotes the starting point of a state machine described by a state chart and the termination symbol denotes the point of termination of a state machine. Figure 626 shows a simple state machine, describing the behavior of a door, including a start symbol and a termination symbol.

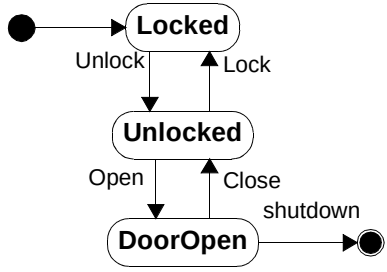


Figure 626: A simple state chart with a start symbol and a termination symbol

Substates

States may be refined into nested diagrams of sub-states, or hierarchical states. The state represents a simplification of more complex behavior expressed in the nested diagram.

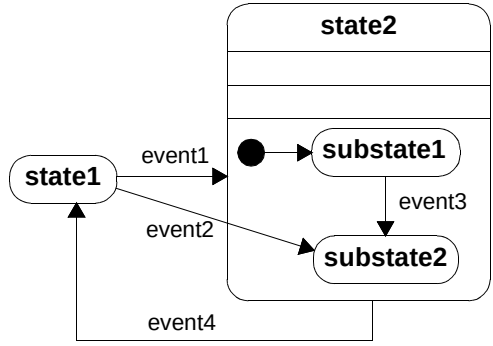


Figure 627: A state chart with states and substates.

State Charts in SOMT

State Charts and SDL

Both State Charts and SDL process graphs are two notations used to express state machines. The two notations have both their advantages. State Charts are good for expressing high level functionality typically used in early activities like analysis. SDL is good for expressing detailed functionality as in design activities. SDL is also a formal notation, with well defined semantics, from which it is possible to generate code.

State Charts in Requirements Analysis

State charts might be used to express high level functionality of a system by focusing on the behavior of the system rather than the behavior of the system's parts. It might also be useful to express the behavior of complex actors that are interacting with the system.

State Charts in System Analysis

In the context of system analysis, state charts are useful to express the behavior of the system parts and how these parts interact dynamically. The architecture of the system is described in a class diagram named Logical Architecture. Using state charts in addition to the class diagram makes it possible to describe the dynamic properties and the behavior of the system or parts of the system. The behavior descriptions should be included in a set of documents called the Object Behavior Diagram which together with the Logical Architecture is part if the Analysis Object Model.

State Charts in System and Object Design

During the activities of System and Object Design, SDL process diagram is the preferred notation for describing the behavior of the system. The product of the Object Design activity should be a complete description of the system which enables code generation. State Charts, due to their weak semantics, may be useful to express a less detailed overview of rather complex behavior expressed in SDL.

Message Sequence Charts

A message sequence chart (MSC) is a high-level description of the message interaction between system components and their environment. A major advantage of the MSC language is its clear and unambiguous graphical layout which immediately gives an intuitive understanding of the described system behavior. The syntax and semantics of MSCs are standardized by ITU-T, as recommendation Z.120 [26].

There are various application areas for MSCs and within the system development process MSCs play a role in nearly all stages, complementing SDL on many respects. MSCs can e.g. be used:

- To define the requirements of a system
- For object oriented analysis and design (object interaction)
- As an overview specification of process communication
- For simulation and consistency check of SDL specifications
- As a basis for automatic generation of SDL skeleton specifications
- As a basis for specification of TTCN test cases
- For documentation

Plain MSC

The most fundamental language constructs of MSCs are *instances* (e.g., entities of SDL systems, blocks, processes and services) and *messages* describing the communication events, see Figure 628.

Another basic language construct is the *condition* symbol which is drawn as a hexagon. A condition describes either a global system state referring to all instances contained in the MSC, or a state referring to a subset of instances (a non-global condition). The minimum subset is a single instance.

An MSC can reference another MSC using an *MSC reference* symbol. (Such a symbol can also reference a High-level MSC, explained later.) This symbol is drawn as a rectangle with rounded corners and has the name of the associated MSC stated inside it. MSC references can for example be used to have one MSC describing an initialization sequence and then reference this MSC from a number of other MSCs.

The reference symbol may not only refer to an MSC but can also contain MSC reference expressions that reference more than one MSC. This construct gives us a very compact MSC representation and it also provides an excellent means for reusability of certain MSCs.

The textual MSC expressions are constructed from the operators alt, par, loop, opt and exc:

- An MSC reference with the keyword **alt** denotes alternative executions of MSC sections. Only one of the alternatives is applicable in an instantiation of the actual sequence.
- The **par** operation defines parallel executions for MSC sections. All events within the parallel MSC sections will be executed (free merge) with the only restriction that the event order within each section must be preserved.
- An MSC reference with a **loop** construct is used for iterations and can have several forms. The most general construct, `loop<n,m>`, where n and m are natural numbers, denotes iteration at least n and most m times. The operands may be replaced by the keyword inf, like in `loop<n,inf>`. This means that the loop will be executed at least n times. If the second operand is omitted, like in `loop<n>`, this will be interpreted as `loop<n,n>`. If both operands are omitted the interpretation will be `loop<1,inf>`.
- The **opt** operation is an operator with one operand only. It is interpreted in the same way as an alt operation where the second operand is an empty MSC.
- An MSC reference where the text starts with **exc** followed by the name of an MSC indicates that the MSC can be aborted at the position of the MSC reference symbol and instead continued with the referenced MSC. If the exception does not occur the events following the exc expression are executed. The exc operator can thus be viewed as an alternative where the second operand is the entire rest of the MSC. MSC references with exceptions are frequently used to indicate exceptional cases when using MSCs to formalize use cases.

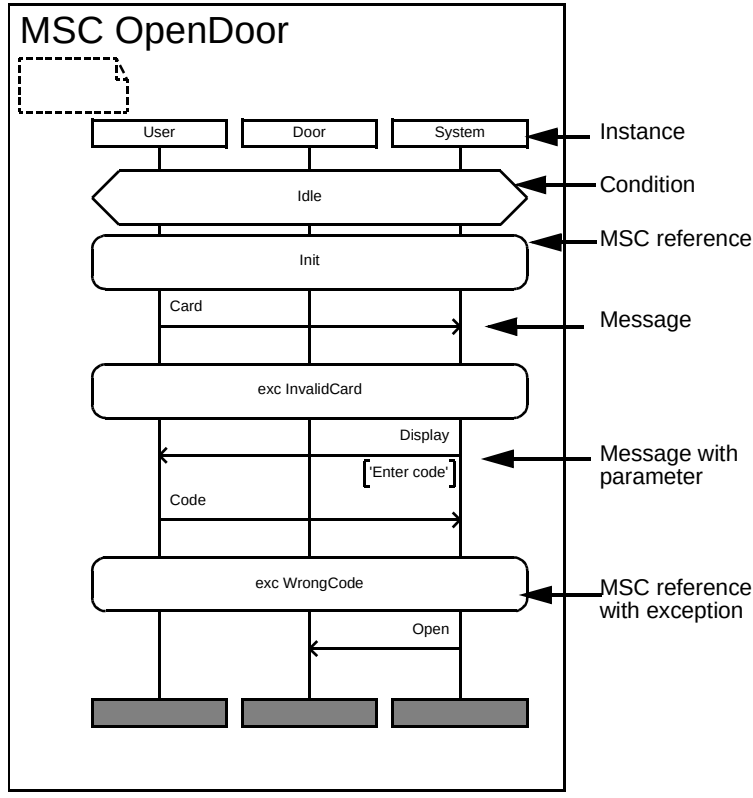


Figure 628: Example of a plain MSC

By means of *inline operator expressions* composition of event structures may be defined inside an MSC. Graphically, the inline expression is described by a rectangle with dashed horizontal lines as separators. The operator keyword **alt**, **loop**, **opt**, **par** or **exc**, placed in the upper left corner, are used with the same meaning as when used together with the MSC reference symbol.

Whether to use inline operator expressions or MSC reference symbols with an operator is a matter of taste. The same things can be expressed with both notations. Using inline expression several scenarios can be expressed in one single diagram, i.e. we just have one single file to handle. If we use MSC reference symbols to express e.g. exceptions and alternatives there is a need for us to handle several files. On the other

hand, the diagram becomes less cluttered if we refer to other MSCs instead of trying to express all possible scenarios in it.

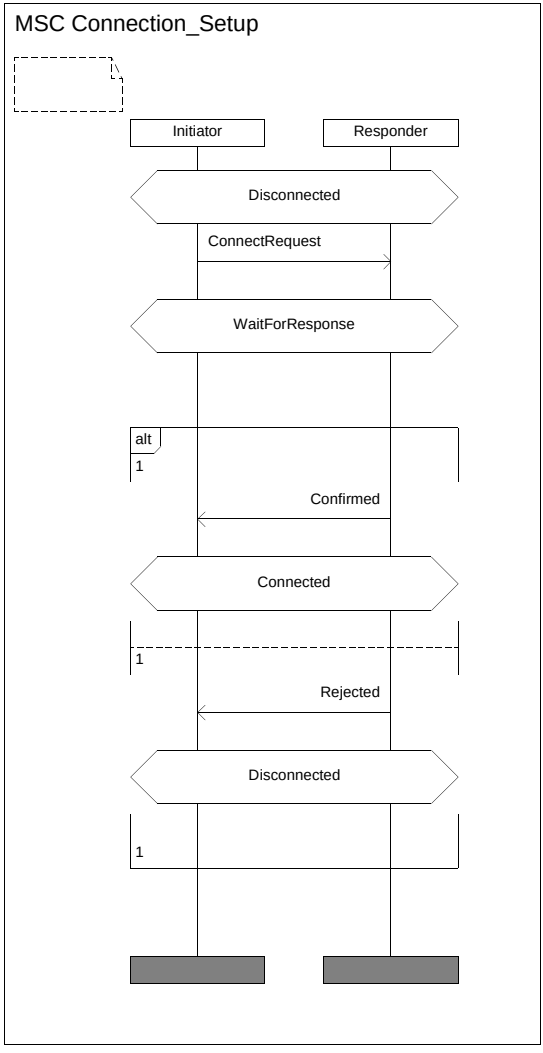


Figure 629: A plain MSC with inline operator expression

For information about more plain MSC concepts, like e.g. timers, please consult the MSC standard Z.120.

HMSC

A high-level MSC (HMSC) provides a means to graphically define how a set of MSCs can be combined. Contrary to plain MSCs, instances and messages are not shown within an HMSC, but it focus completely on the composition aspects. You can get a picture of how an HMSC works in practice by comparing it with a road map. HMSCs, like normal road maps, may easily become quite complex if they are not structured in any way. Fortunately, HMSCs can be hierarchically structured, i.e. it is possible to refine HMSCs by other HMSCs. The power of the MSC language is considerably improved with the new concepts introduced with HMSCs. It is e.g. much easier to specify a main scenario together with all accompanying exceptions.

An HMSC is a directed graph where each node is either (see Figure 630):

- A start symbol which denotes the start of an HMSC (there is exactly one start symbol in each HMSC).
- A stop symbol which denotes the end of an HMSC.
- An MSC reference used to point out another (H)MSC diagram which defines the meaning of the reference. The reference construct in HMSC can thus be seen as a placeholder for an MSC diagram or another HMSC diagram. Like a reference symbol in a plain MSC, an MSC reference symbol in an HMSC can contain references to several (H)MSCs through using MSC reference expressions and the operators alt, par, loop, opt and exc.
- A condition symbol is used to set restrictions on how adjacent referenced MSCs can be constructed. An HMSC condition immediately preceding an MSC reference has to agree with the (global) initial condition of the referenced MSC according to name identification. All conditions on HMSC level are considered to be global, they refer to all instances contained in the MSC. They can be used to guard the composition of MSCs described by HMSCs.
- A connection point which denotes that two crossing lines are actually connected. This symbol has no semantic meaning but is introduced only to simplify the layout of the HMSC.

Flow lines connect the nodes in the HMSC and they indicate the sequencing that is possible among the nodes in the HMSC. If there is more than one outgoing flow line from a node this indicates an alternative.

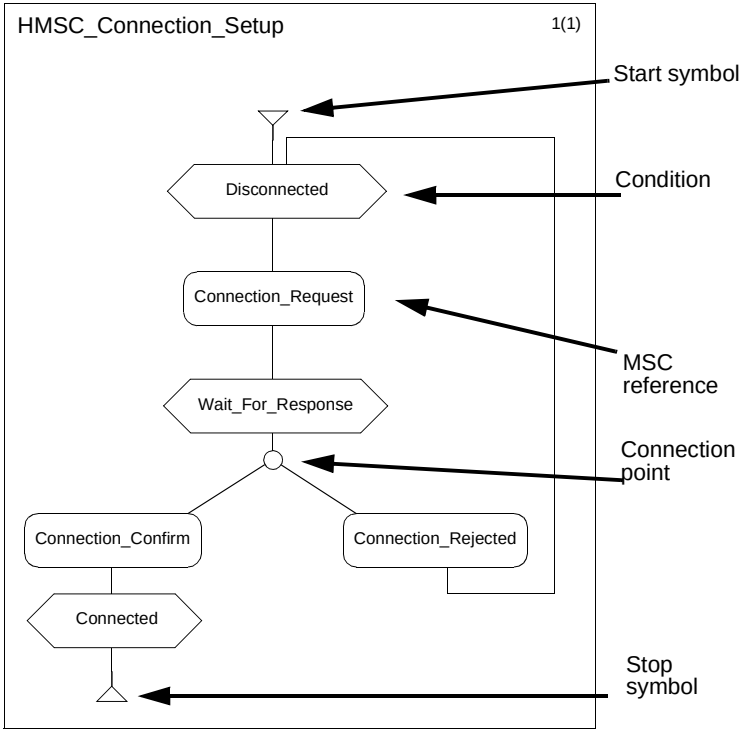


Figure 630: Example of an HMSC

SDL

SDL (Specification and Description Language) is mainly used for specifying behavior of real-time systems. This section provides a small overview of the language. For further reading, we recommend [27], [28] and [29] as useful textbooks about SDL. The SDL language is standardized by ITU-T, as recommendation Z.100 [24].

An SDL system consists of the following components:

- Structure
  - Hierarchical decomposition with *system*, *block*, and *process* as the main building blocks
  - Type hierarchies: inheritance, generalization and specialization are supported for hierarchical building blocks
- Communication
  - Asynchronous *signals* with optional signal parameters
  - *Remote procedure* calls for synchronous communication
- Behavior
  - *Processes*
- Data
  - *Abstract data types* that can be inherited, generalized and specialized
  - ASN.1 data types according to Z.105 [25]. See "ASN.1" on page 3836
- Modularization
  - Components like *block/process types*, *data types*, and *signals* can be placed into *packages* that can be imported into a *system*, enabling separate development.

Structure

Figure 631 shows the hierarchical levels in SDL: *system*, *block*, *process*, *procedure* and *service*.

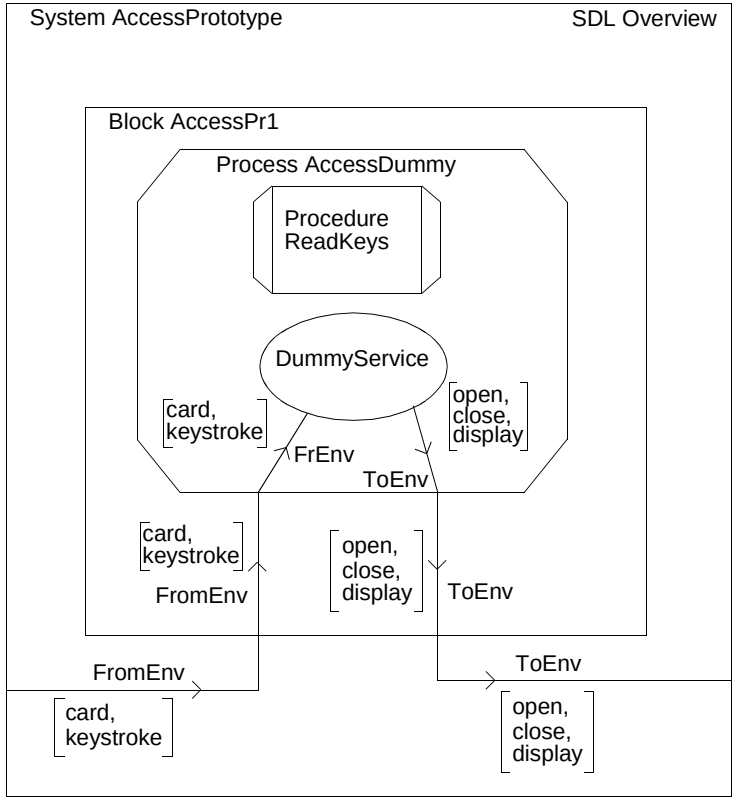


Figure 631: The hierarchical decomposition of an SDL specification

A *system* must contain at least one *block* and a *block* must contain at least one *process*. *Services* can exist within a process, being executed one at a time, controlled by the received signals. Thus the incoming signal sets of services in one process must be disjoint. Using *procedures* is a way to structure the information within processes and services. Processes, services and procedures are described by a flow chart-alike notation, see "Behavior" on page 3827.

The static structure of a *system* is defined in terms of *blocks*. Blocks communicate by means of *channels*.

The dynamic structure of an SDL *system* consists of a set of *processes* that run in parallel. A *process* is a finite state machine extended with data. *Processes* are independent of each other and communicate with discrete *signals* by means of *signal routes*.

Communication

Since SDL does not allow any use of global data, all information that has to be exchanged must be sent along with *signals* between processes, or between processes and environment. Signals are sent asynchronously, i.e. the sending process continues executing without waiting for an acknowledgment from the receiving process.

Signals travel through *channels* between blocks, and from one process to another via *signal routes*. See Figure 631.

Synchronous communication is possible via a shorthand, *remote procedure call*. This shorthand is transformed to signal sending with an extra signal for the acknowledgment. Remote procedures are often used when a process wants to offer services to other processes.

Behavior

The dynamic behavior in an SDL system is described in the processes. Processes in SDL can be created at system start or created and terminated dynamically at run time. More than one instance of a process can exist. Each instance has a unique *process identifier* (PId). This makes it possible to send signals to individual instances of a process. The concept of processes and process instances that work autonomously and concurrently makes SDL appropriate for distributed applications.

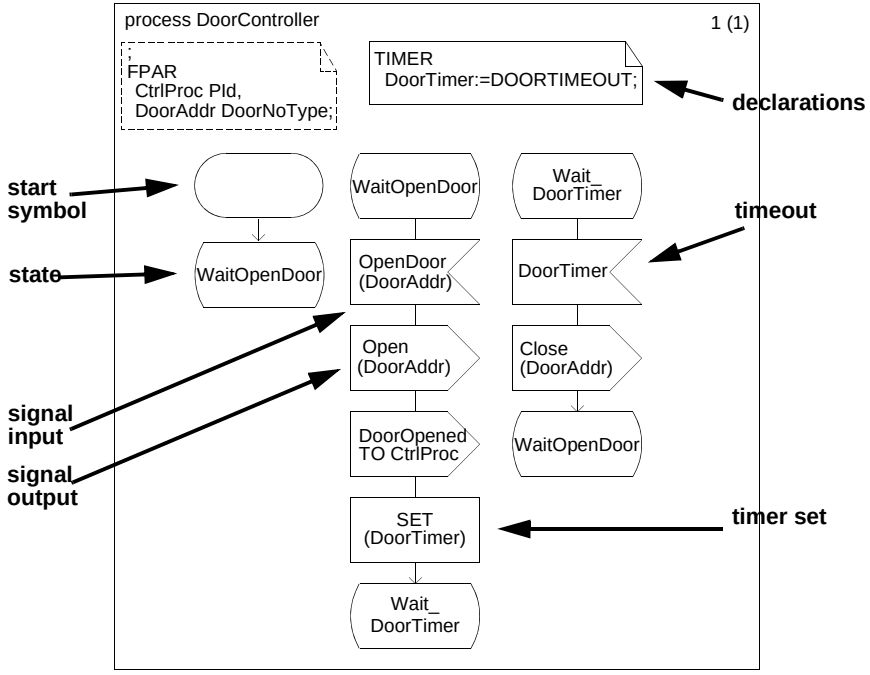


Figure 632: Behavior described by an SDL process

A process must have one start symbol. Since a process is a state machine, a transition between two states is made only after a signal has been received. If there are no incoming signals the process is inactive in a state. In SDL a transition takes no time. To be able to model time, and to set time restrictions, there is a timer concept. Each process has its own set of timers that can be set to expire on different durations.

Data

The set of predefined sorts in SDL makes it possible to work with data in SDL in a traditional way:

- Integer
- Real
- Natural
- Boolean
- Character
- Duration
- Time
- Charstring
- PId

More complex data sorts can be created by using *arrays*, *strings* and *structs*.

Abstract Data Types in SDL can be used for more than representing data, e.g:

- Hide data manipulation
- Hide algorithmic parts of a specification
- Create an interface to external routines

Data manipulation is hidden in *operators*. For a more thorough description on how to use complex data structures with operators in practice, please see [32].

Structural Typing Concepts

The object-oriented concepts of SDL give you powerful tools for structuring and reuse. The concept is based on type definitions. All structural building blocks can be typed: *system type*, *block type*, *process type* and *service type*. An exception is the *procedure* that is a type in its original form.

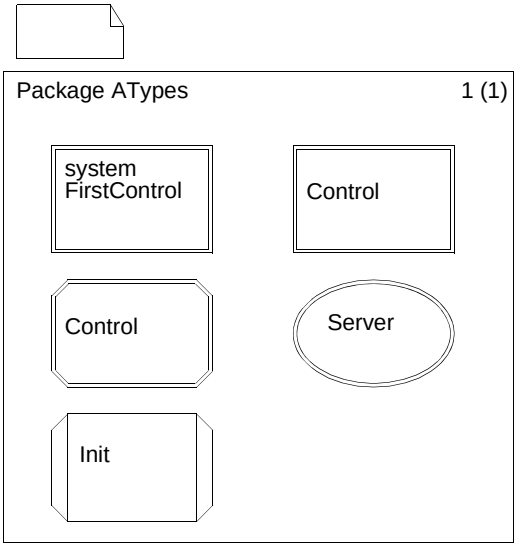


Figure 633: Package with type definitions

Type definitions may be placed outside the system in *packages*. *Packages* can be seen as libraries of frequently used functions. The structural typing concepts are shown in Figure 633. All types can inherit from other types of the same kind.

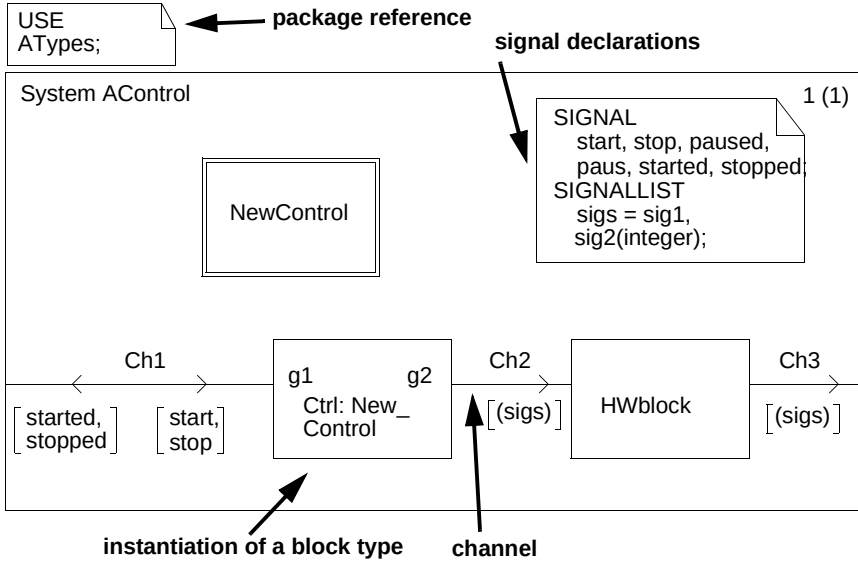


Figure 634: A system diagram with block instantiation and communication

One of the major benefits of using an object oriented language is the possibility to create new objects by adding new properties to existing objects, or to redefine properties of existing objects. This is what is commonly referred to as *specialization*.

In SDL-92, specialization of types can be accomplished in two ways:

- A subtype may add properties not defined in the supertype. One may, for example, add new transitions to a process type, add new processes to a block type, etc.
- A subtype may redefine virtual types and virtual transitions defined in the supertype. It is possible to redefine the contents of a transition in a process type, to redefine the contents/structure of a block type, etc.

Figure 634 and Figure 635 describe adding and redefining properties in a system and in a block type while Figure 636 describes the same features in a process type diagram.

To be able to instantiate a type regardless the context (by means of channels), a special concept is needed: *gates*.

Since a channel always has to be connected to a signal route and the connection mechanism lies inside the process, a gate is necessary since it is a way to specify the connection in a transparent manner.

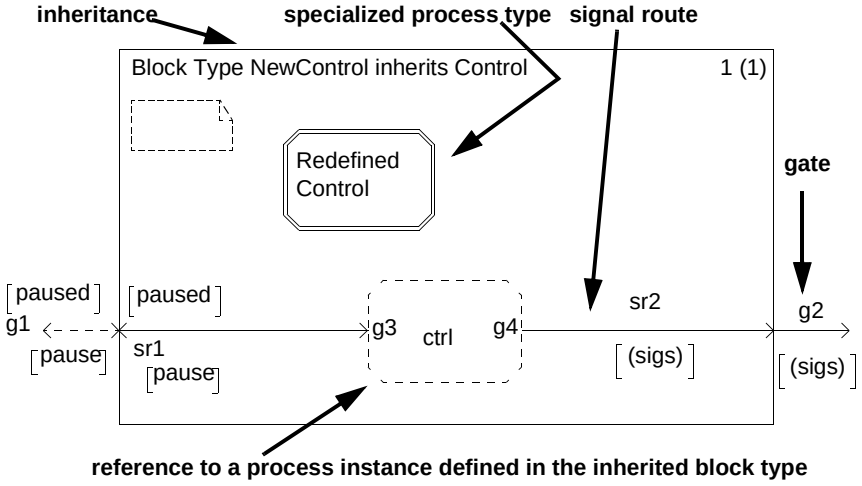


Figure 635: An inherited block type diagram with process specialization, instantiation and communication

In the inherited block type Control, the process ctrl is an instance of the process type Control.

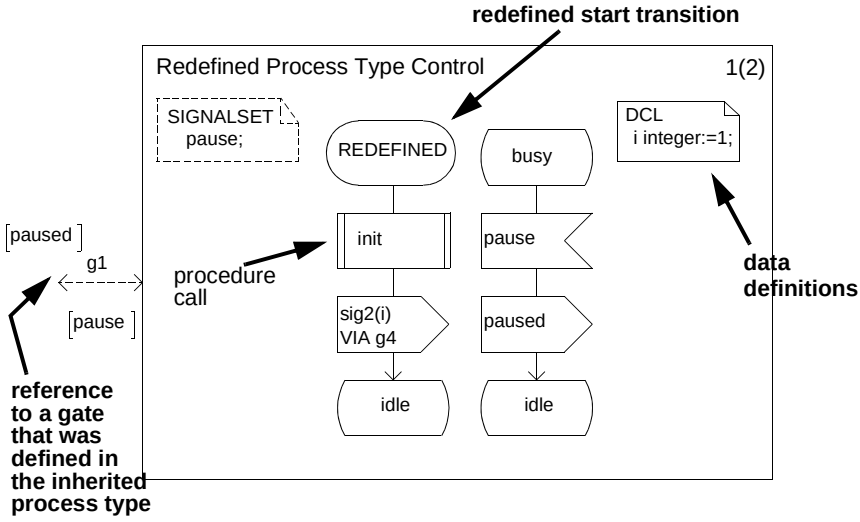


Figure 636: A specialized process type with added signals, a new transition and a redefined transition

Graphical and Textual Notation

The SDL language supports two notations that are equivalent. Beside the graphical representation (SDL/GR), a textual phrase representation (SDL/PR) is standardized.

TTCN

TTCN (Tree and Tabular Combined Notation) is a special purpose notation to describe test suites [34]. TTCN is a language standardized by ISO for the specification of tests for communicating systems. TTCN has been developed within the framework of standardized conformance testing (ISO/IEC 9646).

With TTCN a test suite is specified. This is a collection of various test cases together with all the declarations and components it needs.

Each test case is described as an event tree. The tree is represented as an indented list in a table. The indentation represents progression with respect to time. See Example 644.

Example 644: A TTCN test case ----------

```
     Behavior Description                Constraint        Verdict
1    DuToEnv? Display                    Enter_Card
2     EnvToDu! Card                      Card1
3      DuToEnv? Display                  Enter_Code
4       EnvToDu! Digit                   digit1
5        EnvToDu! Digit                  digit3
6         EnvToDu! Digit                 digit5
7          EnvToDu! Digit                digit7
8           DuToEnv? Unlock              nopar
9            EnvToDu! Open               nopar
10            DuToEnv? Display           Please_Enter
11             EnvToDu! Close            nopar
12              DuToEnv? Lock            nopar
13               DuToEnv? Display        Enter_Card        PASS
14         EnvToDu! Digit                digit1
15          DuToEnv? Display             WrongCode
16           DuToEnv? Display            Enter_Card        PASS
```

Each line consists of a line number, a statement, a constraint reference and a mandatory verdict. A statement can be:

- An event
- An action
- A qualifier

The event statements are statements that can be successful dependent on the occurrence of a certain event, either:

- Receive (represented by a “?”)
- Otherwise
- Timeout

The action statements will always execute and will therefore always be successful:

- send (represented by a “!”)
- implicit_send
- assignment_list
- timer_operation
- goto

A qualifier is simply an expression that must be true if an event should match or an action should be performed.

In TTCN, the communication is asynchronous. The *implementation under test*, IUT, communicates with the environment via *points of control and observation*, PCOs. The interaction occurs at PCOs and are described by *protocol data units*, PDUs, embedded in *abstract service primitives*, ASPs.

At line 1 in s above, the ASP `Display` occurs at the PCO `DuToEnv`. The constraint `Enter_Card` determines exactly which ASP value is to be received.

The leaves in the tree are usually assigned a verdict.

ASN.1

ASN.1, described in [23], stands for Abstract Syntax Notation One. ASN.1 is a language for the specification of data types and values. ASN.1 is very popular for the specification of data in telecommunication protocols and services, especially in higher (i.e. application oriented) layers. Many telecommunication standards are based on ASN.1. Also TTCN is based on ASN.1. An example of an ASN.1 module is shown below.

Example 645: An example of an ASN.1 module

```
ProtocolData DEFINITIONS ::=
BEGIN
- - contains data definitions for an example
protocol

CheckSum ::= INTEGER (0..65535)

DataField ::= OCTET STRING (SIZE (0..56))

PDU ::= SEQUENCE {
        sequenceNr INTEGER (0..255),
        dataField DataField,
        checksum CheckSum OPTIONAL }

END
```

A strong point of ASN.1 is that there are encoding rules that define how an ASN.1 data value is encoded to bits, the most well-known being the Basic Encoding Rules. From an ASN.1 data type definition, functions can be automatically generated that take care of the coding and decoding.

An ASN.1 definition can be imported into SDL as if it was a package. When an ASN.1 data type is imported into SDL, automatically a set of operators that is defined in Z.105 [25] is available for that type.

It is recommended to use ASN.1 for the specification of parameters of signals to/from the environment of the SDL system, especially when encoding rules are to be applied on such signals, or when a TTCN test suite is to be developed. In the latter case the ASN.1 definitions can be directly reused in the TTCN test suite.

Chapter

71 *Requirements Analysis*

This chapter gives a thorough description of the different models in the requirements analysis activity as well as some guidelines on how to create these models. A recommendation on consistency rules that are relevant for the models in this activity is also included.

The chapter requires that you are at least reasonable familiar with the concepts concerning object models as well as the concepts concerning MSC diagrams.

Requirements Analysis Overview

The purpose of the requirements analysis is to establish an understanding of the application domain and to capture, formalize, analyze and validate the user requirements on the system to be built. For this purpose the system is viewed as a black-box and only the objects and concepts visible on the system boundary and outside the system are modeled.

The input to this activity can of course be very different depending on the application, from an extensive requirements specification provided by a customer to some more or less vague ideas not documented on paper. In any case the result of the activity should be the same: An understanding of the problem domain that the system is to be operating within and an understanding of the role the system is to fulfill in this environment, i.e. the requirements on the system to be built.

In many cases the requirements analysis can be divided into two activities:

- A problem analysis
- A system requirements analysis

The problem analysis concentrates on understanding the problem domain the system is going to operate in, without any reference to the system itself. The system requirements analysis focuses directly on the functional requirements on the system, viewed as a black box.

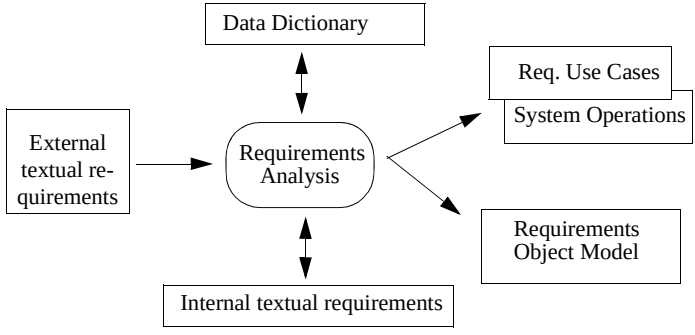


Figure 637: Overview of the requirements analysis activity

An overview of the requirements analysis activity is depicted in Figure 637. As can be seen a number of models are used:

- Textual requirements model as described in "Textual Requirements" on page 3840. Some requirements are supplied as input to the activity, others may be produced within the activity.
- Data dictionary, further described in "Data Dictionary" on page 3841.
- Use case model, described by text, plain MSCs, or plain MSCs in combination with HMSCs. The use case model shows the system and the actors interacting with the system. It is described in "Use Cases" on page 3843.
- Requirements object model using object model notation. This model includes both a problem domain model and context diagrams showing the system and the external actors that interact with the system. The requirements object model is described in "Requirements Object Model" on page 3851.
- System operations model, see "System Operations" on page 3856.

The requirements analysis is a highly iterative process where the different tasks creating the different models are iterated over and over again but the following outline can give some guideline on how to structure the work:

1. Study any textual requirements that are provided as input and other sources of information that are available, e.g. read text books about the problem domain and talk to potential users of the system. This may also include rewriting (or creating from scratch) the textual requirements to make them more complete or structured.
2. While performing 1 above create a first version of the data dictionary, including lists of actors, use cases and important problem domain concepts.
3. Create a first version of the requirements object model:
   - Use context diagrams to give an overview of the actors and the system.
   - Use an information model to describe the problem domain.

4. Create a first version of the use case model.
   - Review and optimize the list of use cases found so far.
   - Describe the details of the most important use cases as text and/or MSCs. If the use cases are complex, HMSCs can be used in combination with plain MSCs. Start with the normal cases and leave the exceptional cases for the moment.
5. Refine and extend the requirements object model and use case model until the use cases and their exceptions cover a sufficient part of the requirements. Add state charts to the requirements object model if the behavior of an object needs to be considered. When needed use system operations to define the details of the interactions between the actors and the system.

Textual Requirements

Conventional textual requirements are in most development projects an important input and result of the requirements analysis. In SOMT, the textual requirements model is simply one or more text documents. It is however important to include them within the scope of SOMT to get a possibility to create implinks from them to the other models in SOMT, and thus make it possible to trace the implementation of the requirements.

The source of the requirements may in some cases be the customers in which case the textual requirements form one of the inputs to the requirements analysis. In other cases the textual requirements are created by the development team as part of the requirements analysis.

Non-functional requirements are an example of a type of requirements that are important to capture, but may be difficult to formulate using the other models in the requirements analysis. The non-functional requirements may express properties about for example:

- Performance issues like response time or number of transactions per second.
- Reliability like the mean time between failure.

This type of requirements are most easily expressed in natural language and entered into the textual requirements model.

It is important to not only write down the requirements but also to analyze them. One aspect of this analysis is to mark all important concepts in the requirements that can be useful later when identifying objects and use cases.

Data Dictionary

The data dictionary is a textual list of all concepts that are defined during the analysis. The purpose of the data dictionary is to define a vocabulary that is common to all the members of the development team and to the customers and users of the system.

It is important to notice that the data dictionary is not a separate list of entities that is unrelated to the other models in the requirements analysis. Rather it can be seen as a different viewpoint on the same set of basic concepts that are used and defined in the use cases and in the requirements object model. For example all domain objects should be added to the data dictionary as soon as they are defined in the object model.

For each entity defined in the data dictionary at least the name of the entity and a brief explanation of the entity must be supplied.

Note that although the data dictionary is created in the requirements analysis it can be used during the entire development process and be updated when new concepts are found also in the other activities. The following Example 646 shows a part of a data dictionary for the access control system.

Example 646: A part of a data dictionary description

```
NOUNS

Access control system - A system to control the
access rights to an office. Unauthorized persons
should not be able to enter without permission.

Card - Each employee working in the office has a
card and a corresponding personal code.

Card reader - The hardware into which the employee
enters the card. The cardreader reads the cardId.

Code - Each employee has a four digit personal code
connected to the card. To enter the office the
employee must type the code on the keypad.
```

Display - The hardware unit by which the system tells the employee what to do.

Door - Employees enter and exit the office by opening a door. The door is always kept locked and unlocks only when the employee enters the office with a valid card and code or when the employee is leaving the office by pressing the exit button. Each door provides a cardreader, keypad and display on the outside of the office and an exit button on the inside.

Keypad - The hardware unit used to type personal codes. The keypad has keys for the digits 0-9.

Employee - The holder of a position in the office. Every employee has a registered card with a personal code to get access to the office.

RELATION PHRASES

Card with code - Each employee in the office has a card with a a personal code.

Code consists of four digits - The personal code that every employee has got, consists of four digits.

Door provides local panel - Each door has a localpanel on the outside of the office. This localpanel is made up of a card reader, a display and a keypad.

Door provides exit button - Each door has an exit button on the inside of the office.

Registerfile contains employee identifications, cardnumbers and codes - The registerfile contains the data relevant to the access control system.

VERB PHRASES

Connection is lost - The connection between a door and the central controller can sometimes fail. In case of broken connection nobody can enter the office. It is, however, possible to leave the office.

Inform employee - The system gives the employee instructions by means of the display.

```
Enter office - A use case which describes the
interaction between an employee and the access
control system when the employee wants to enter the
office.

Exit office - A use case which describes the
interaction between an employee and the access
control system when the employee wants to exit the
office.
Validate card - The central controller validates a
card with respect to info in the registerfile.

Validate code - The central controller validates the
correctness of a code with respect to info in the
registerfile.
-------------------------------------------------------
```

In Example 646, the items in data dictionary are categorized into nouns, relation phrases and verb phrases. By categorizing the items, it may be easier to find the objects, attributes, operators, actors and use cases and also the relations between these entities in each model.

Use Cases

The most important problem to tackle when designing a new system is not to verify the correctness of the system or to get an optimized implementation. None of this matters if the system *does not solve the right problem*. User-centered requirements analysis using use cases is an approach to capture requirements from the user's point of view. The intention is of course that the users will be able to understand and validate the use cases, and thus confirm that they indeed define the right system to be built.

Consider a system that controls the access to a building, allowing users to enter the building through doors that can be opened by entering a card and a code into a card reader. Some typical use cases for this system may be:

- "Register a new user"
- "Open door"

An important term when discussing use cases is the notion of an "actor". An actor is an outside entity that interacts with the system. An actor can be a human being, a hardware unit, a computer program or anything else that can communicate with the system. Usually there is also made a distinction between the individual users of a system and the actors. An actor is not supposed to be an individual user, but rather represents one of

the different “roles” individual users can play when interacting with the system.

Each use case describes essentially the possible sequences of events that take place when one or more actors interact with the system in order to fulfill the purpose of the use case. Note that the use case does not define one specific sequence of events, but rather a set of possible sequences.

A use case is thus simply a description, in one format or another, of a certain way to use the system. It has been found to be a very efficient way to capture a user’s view of the system and the concept of use cases is now used in a number of object-oriented methods. The version used in SOMT is mainly a combination of the original use cases as described in [19] and the OMT version [20]. A difference is that, as will be seen below, SOMT puts some more effort in the formalization of the use cases to be able to use them for verification purposes during the object design.

In SOMT the use case model is in practise composed of three parts:

- A list of actors
- A list of use cases
- The collection of use case descriptions

The list of actors should describe all actors that have been identified, why they use the system (or which service they provide to the system) and what their responsibilities are.

The list of use cases just gives a one-sentence description of each use case.

The lists of actors and use cases can either be separate documents or be a part of the data dictionary. The lists are particularly useful in the beginning of the requirements analysis when trying to identify the most important use cases and actors.

Two different formats are used to describe use cases in SOMT. One purely textual format and one format that uses MSC diagrams to define the use cases. Depending on the application (and the users) one or both of the notations can be used in a specific project. If the chosen approach is to describe the use cases by means of MSCs and they turn out to be very complex, it is possible to use a combination of plain MSCs and HMSCs to describe them. Most of the things that can be expressed with

HMSCs can also be expressed by plain MSCs, so it is a matter of taste if you want to use HMSCs or not. HMSCs have certain advantages, especially if the system is complex:

- They give a good overview of what is happening in the system and thereby facilitate understanding of the system behavior.
- They make it easier to maintain the use case model.

The choice of notation for use cases is a matter of personal taste and the application at hand. There are four different possibilities:

- Use only textual use cases.
- Use only MSC use cases.
- Give a brief textual description of the use case. Then give a complete formal MSC definition of the use case.
- Give a complete detailed textual description of the use case, covering all exceptions etc. Then use MSCs to exemplify some of the more important cases.

Textual Use Cases

The textual format consists essentially of natural language text structured into a number of fields and is easiest introduced with an example. Consider the *Enter building* use case for the access control system that controls the doors of a building. This use case is depicted in Example 647.

Example 647: Textual description of the *Enter building* use case––––

```
Use case name: Enter building
Actors: Regular user, Door
Preconditions: 'Enter card' is displayed and doors
are closed and locked
Postconditions: Same as preconditions
Description: A user enters a card into the
cardreader. 'Enter code' is displayed on the
display. The user enters his code (4 digits) using
the keyboard. The door is unlocked and 'Door
unlocked. Please enter.' is displayed. The user
opens the door, enters the building and closes the
door again. the door is locked again and 'enter
card' is displayed.
```

```
Exceptions: - If the user enters the wrong code then
'Wrong code' is flashed for 3 seconds and then
'enter card' is displayed.
- If the user does not open the door after it has
been unlocked, then the door is locked again after
30 seconds and 'Enter card' is displayed
------------------------------------------------------------
```

As can be seen in the example, a use case has the following fields:

- Name: The name of the use case.
- Actors: A list of the actors involved in the use case.
- Preconditions: A list of properties that must be true for this use case to take place.
- Postconditions: A list of properties that are true when the use case is finished.
- Description: A textual description of the "normal" sequence of events that describe the interaction between the actors and the system.
- Exceptions: A list of exceptional interactions that complement the "normal" flow of events described in the *Description* field. If an exception leads to different postcondition properties compared to the "normal" sequence, this should be noted.

As with the textual requirements, it is important to analyze the textual use cases and clearly mark all important concepts in order to be able to use them when identifying classes and when performing consistency checks between the different models.

Message Sequence Charts

The second notation for use cases used in SOMT is Message Sequence Charts (MSCs). An MSC is a diagram that shows a number of communicating entities (called instances) and the messages that they exchange. Two MSCs that correspond to the "Enter building" use case are shown in Figure 638 and Figure 639. Figure 638 describes the normal case and Figure 639 describes one of the exceptions.

A comment to this use case: There is always a choice of how much to show in a use case. In this particular example there is of course a choice whether to show the interaction between the *User* and the *Door* that

does not involve the system. There is of course such an interaction where the user opens and closes the door. What to do depends on the purpose of the use case. Is the purpose mainly to describe and understand the problem domain or is the purpose to define the precise requirements on the system? In this use case we have assumed that the purpose is to define the requirements so the interaction that does not directly involve the system is left out of the MSC.

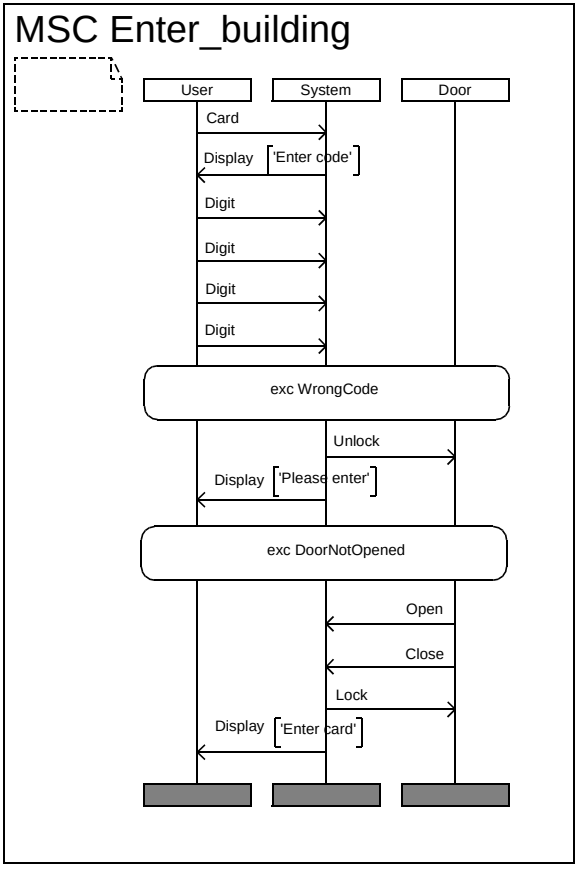


Figure 638: The “Enter_building” use case

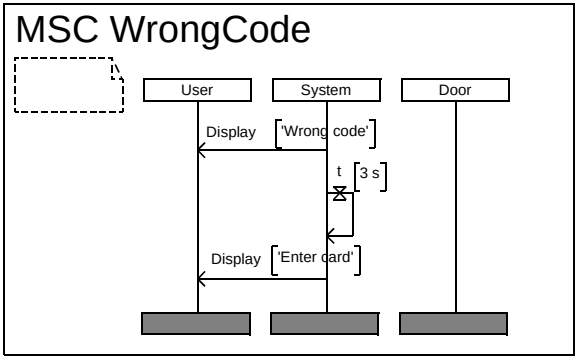


Figure 639: The exception "WrongCode" for the "Enter_building" use case

Identifying Use Cases

Often some use cases are found very easily from the purpose of the system, but in some cases it is more difficult. One strategy that works well, both as a means to find use cases and as a means to check that all important use cases have been found, is:

1. Identify the actors.
2. Identify the use cases needed by each actor.

These two tasks are discussed in the following sections.

Finding Actors

By reading the textual requirements model or discussing with customers, candidates for actors can be found through the answers to the following questions:

- Which user groups need help from the system to perform a task?
- Which user groups are needed by the system in order to perform its functions? These functions can be both:
  - Main functions
  - Secondary functions, such as system maintenance and administration

- Which are the external hardware or other systems (if any) that use the system or are being used by the system in order to perform a task?

Now, we have a set of possible actors. From this, we have to decide if these candidates really are actors or if they are parts of the (software) system. By doing this, we identify the *boundaries* of the system. This step does not provide any final solutions and decisions taken here might be modified later.

If the actor represents a user, perhaps one, single user, try to keep the level of abstraction that comes with the actor concept. It might be the case that this specific user can perform some tasks that can be performed by other users and some unique tasks. Therefore, try to distinguish between the roles that the users can play, i.e. the actors, and the users themselves. See Figure 640.

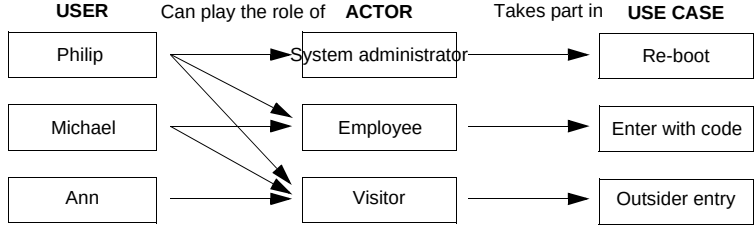


Figure 640: Example of difference between users and actors

Document all actors in a list, describe each actor by a name and describe briefly its role when interacting with the system. The description should contain the actors' responsibilities and the way the actor uses the system. During the process of identifying the actors these descriptions are most conveniently kept in the textual document that lists all actors. This can either be a separate document or a part of the data dictionary.

The set of actors may be due to changes after having started to describe the use cases. Especially the MSC diagrams are useful to pinpoint the problem of determining the actors. In practice this means that the activity of identifying use cases by defining actors and use cases should be performed iteratively.

Finding Use Cases

When you have defined a set of actors, it is time to describe the way they interact with the system.

Steps:

1. For each actor, find the tasks and functions that he should be able to perform or tasks which the system needs the actor to perform. Search the textual requirements for verb phrases, these are possible candidates for use cases.
   - If possible, make the use cases as complete as possible, i.e. make *one* use case of a complex function rather than splitting it up into several use cases for each sub-function.
   - To begin with, concentrate on the normal cases. Leave the exceptional cases until a later stage.
2. Name the use cases and enter them in a textual list of use cases.
   - The names should be informative and be accompanied with general descriptions of the use case functionality.
   - The naming should be done with care, the description of the use case should be descriptive and consistent. Example: the use case that describes when a person leaves deposit items to a recycling machine could either be named *Receive deposit items* or *Returning deposit items*, but the latter is preferable since it is usually better to give names that reflect the users point of view rather than the systems.
   - From the list of use cases, try to discard unnecessary use cases. A use case should represent a set of events that leads to a clear goal (or in some cases several distinct goals that could be alternative) for the actor or for the system.
3. Describe the use cases using the textual use case format and/or give a formal description of the use cases using MSCs (combined with HMSCs if necessary). Use the terms in the requirements object model and data dictionary as much as possible. Focus to begin with on the normal cases.
4. Refine the use cases by examining the exceptional cases that are possible for each use case.

Now, we have got a view of possible candidates for use cases. It is not sure that all of these need to be described in separate use cases; some of them may be modeled as exceptions to other use cases. Consider what the actor *wants* to do!

While finding (or specifying) the use cases, it might be the case that you have to make changes to your set of actors. All actor changes should be updated in the textual list of actors and use cases. The changes should be carried out with care, since changes to the set of actors affect the use cases as well.

Requirements Object Model

The purpose of the requirements object model is to document all the concepts found during the requirements analysis and the relations between these concepts. The benefits of establishing this type of model are obvious:

- The developer and users get a common medium that can be used to check that they have a common understanding of the problem.
- It can to a large extent be reused in the system analysis as a foundation of the system object model.
- It is invaluable when new members of the development team are introduced into the problem domain.

As we have already seen there are different categories of concepts that can be described in the requirements object model. The two major types of requirements object model diagrams show either:

- The logical structure of the data and information
- The environment and context of the system, including the actors that use the system.

Finding the Objects

The classical question when discussing object models is: How do we find the objects?

The final answer to this question is yet to be found, but some obvious sources of information are:

- The use cases
- Textual requirements

The use cases are helpful in more than one way. They directly define the actors that interact with the system and these are of course obvious object candidates. They also describe what is to be entered into the system and what will come out of the system. This may be physical entities like the card in the access control system or abstract data like a data unit in a telecommunication protocol. In either case the entities that are transported in to or out from the system are likely candidates for the requirements object model.

If there exist textual requirements specifications a classical way to find the objects that may be useful is to study the requirements and note all nouns that are used. If a particular substantive is used in many places it may represent a concept that is worth including in the requirements object model.

Other possible sources of information that can be helpful in finding the objects include:

- General domain knowledge from text books or experts
- The physical environment the system will operate in

Finding Relations

After we think that we have found a sufficient number of objects, we want to relate these objects, or more generally, the classes. There are three different kinds of relations:

- Aggregation – describing a “consists of” relation
- Generalization or inheritance – describing a relation where one of the classes are generalized from the other class(es)
- Association – describing how different classes (that are not closely related by aggregation or generalization) are related by means of information exchange

Relations can be found by

- Searching the textual requirements for “relation phrases”, for example: “card with code”, “the central controller has access to the register file” and “each local panel consists of a display, a keypad, and a card reader”
- Looking in the textual use case description

In order to increase to readability of the model, name the associations. If needed, you can also attach role names to each class in an association. Generalization and aggregation relations can also be named.

Finding Attributes and Operations

Closely related to the associations are the *attributes*. Attributes are entities (that could be classes of their own) that are considered to be individual for a class.For example: name, address and phone number could be three different attributes of the class person. This information can be found in:

- The textual requirements
- The use cases

Sometimes it is not easy to decide if an entity should be an attribute of a class or if it should be a class of its own and have an association to the other class. If the independent existence of a property is important, then it should be a class. Consider also possible future changes and extensions, this might also give a reason for making the entity a class of its own. Example: A card in the access control system could change owner after a reorganization.

Operations describe functionality of a class. Operations are often used to modify the attributes of a class. If an operator has features of its own (attributes that do not have to be known for the whole class), then it could be modeled as an individual class.

Operations can be found

- By looking at the system operations and distributing the responsibilities to several classes
- In the textual requirements
- In the textual use cases

Information Modeling

The requirements object model forms a description of the problem domain in a graphical way using object model diagrams. The purpose of the requirements object model is to give an overview of the concepts used by experts in the problem domain when discussing various aspects of the problem. Notice that the object models give the overview of the concepts, more details should be defined in the data dictionary. In the

requirements object model the focus is on classes and associations between classes.

Another way to view the requirements object model is that it at least should describe all concepts that are visible on the “outside” of a system. Note that this does not only include physical entities that a user can see but also the knowledge the user must have to be able to use the system.

As an example of a small requirements object model consider Figure 641 that shows a requirements object model for an access control system that describe the different kinds of persons that are of interest for the application area, how they relate to cards and codes and the structure of an office building.

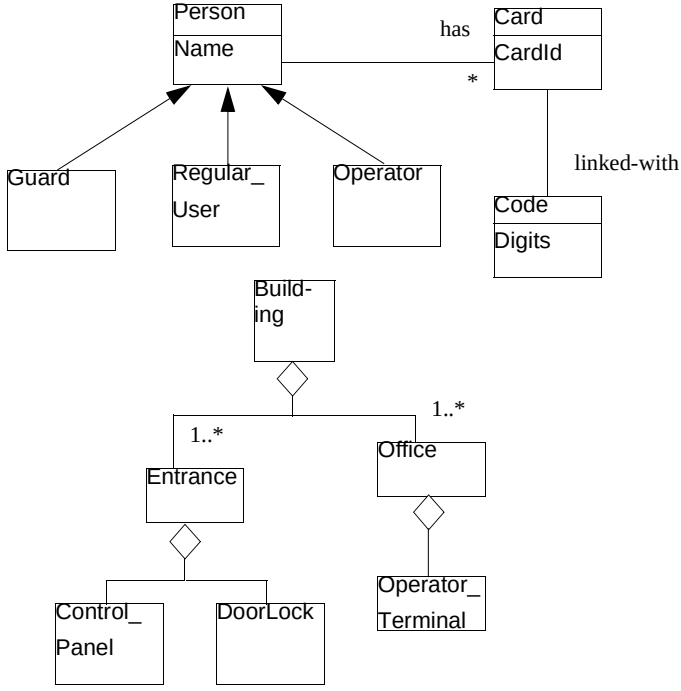


Figure 641: A requirements object model for an access control system

Context Diagrams

Context diagrams are intended to give a static overview of how the system will interact with its environment. This is accomplished by showing in a class diagram (if needed exemplified in instance diagrams) what external actors exist that will interact with the system. The context diagrams are obviously very closely linked with the use cases since they show all types of actors that are defined in the use cases. Actually, one of the major benefits with the context diagrams is thus that they in one (or a few) diagrams capture the static information that otherwise is hidden in the use cases. As an example, consider Figure 642 that shows a simple context diagram for the access control system.

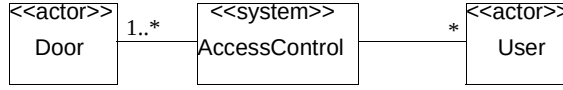


Figure 642: A context diagram for an access control system

Instance diagrams can be used to show specific configurations or examples as in Figure 643 which show a situation where there are three doors and two users.

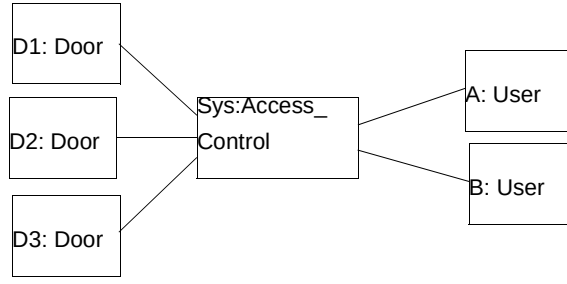


Figure 643: A context diagram using object instances instead of classes to show an application example

An important aspect of the context diagrams is that they show where the borderline between the system and the environment is, event though it is shown in an abstract fashion.

Modeling Behavior

In some cases it is useful to give a description of the internal behavior of the most important objects that appear in the object model. Due to the simplicity of the notation State Charts is the preferred notation in the analysis activities. As an example consider the DoorLock object in Figure 641. A description of the behavior of this object is shown in Figure 644.

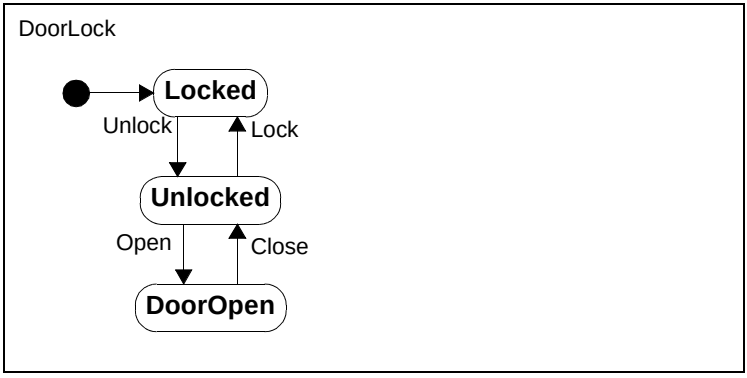


Figure 644: An SDL state machine describing the behavior of the DoorLock object

System Operations

In the use cases a number of events can be found that form the elementary communication means that is used when the system and the actors interact. For example in the “Enter building” use case in Figure 638 there are several events, e.g. “enter card” and “enter digit”. It is important to understand and document the events that are used in the use cases, since they define the interface between the system and its environment. This can be done in the requirements object model and in the data dictionary, but sometimes a more detailed definition is useful. This is the purpose of the *system operations*, a concept that originates from the Fusion method [30].

A system operation is a definition of what that system must do to handle an event. It defines declaratively the behavior of the system as a response to an event in terms of the changes of state and events that are output or returned. System operations are in SOMT defined using sche-

mata containing structured text. As an example consider the system operation in Example 648 that defines the *Enter card* operation.

Example 648: A system operation schemata for the enter card operation ----------

```
Operation: EnterCard
Responsibilites: Informs the system of the fact that
a card has been entered into one of the card readers
Inputs: Card identificatio Card reader
identification
Returns: Enter code' is displayed
Modified objects: A 'card' object,a 'card reader'
object
Preconditions: The door must be closed
Postconditions: The card identification is stored in
a 'card' object associated with the 'cardReader'
object with an id equal to the card reader
identification that was an input to the operation
```

The different rows in the schemata has the following meaning:

- Operation: The name of the system operation (equal to the name of the event that it handles).
- Responsibilities: A short description of the operation.
- Inputs: The data that is supplied as parameters.
- Returns: The events that are returned to the outside of the system as a result of the operation.
- Modified objects: The internal objects that are changed by the operation.
- Preconditions: Predicates that define when definition of the operation given in this schemata is valid.
- Postconditions: The postconditions for the operation define how the state of the system has changed due to the execution of the operation.

Note that in general more than one schemata is needed for each system operation: one schemata is needed for each possible variant of the preconditions.

The criteria for choosing to use the system operations in addition to the use cases are based on the estimated complexity of the opera-

tions/events in the use cases. If the events are simple there is no need for a more detailed description of them. If the events are complex, e.g. when they are parametrized with complex data structures or involve some complex algorithm, then system operations are useful as a means to define the details.

Consistency Checks

This section contains some consistency checks applicable to the models in the requirements analysis. The following list should be viewed as suggestions and must be adapted to the way the requirements analysis is performed in a particular project:

- Check the use cases and requirements object model with actual users.
- Reread the textual requirements and check that all important concepts are clearly marked as such.
- Reread the textual use cases and check that all important concepts, like actors, are marked as such.
- Check that all important concepts in the textual requirements model and in the use cases, and all entities in the requirements object model are added to the data dictionary.
- Check that all important concepts in the textual requirements are modeled by the requirements object model.
- Check that all actors in the use cases are modeled in the context diagram(s) and vice versa.
- Check that all functional requirements in the textual requirements model are modeled by the use cases.
- Check that all complex events in the use cases are described by system operations.
- Check that the requirements object model follows the syntactic/semantic rules for object models.
- Check that the MSC use cases follow the syntactic/semantic rules for MSCs.

Summary

The requirements analysis is an activity that is focused on understanding and documenting the problem domain and the requirements on the system. The major models produced are:

- A textual requirements model
- A use case model, where the different ways a system is to be used are defined
- A requirements object model, i.e. a description of the objects needed to understand the problem domain and external requirements
- System operations
- The data dictionary, which is a list of concepts used in the problem domain and requirements

Chapter 71 Requirements Analysis

Chapter

72 *System Analysis*

This chapter gives a thorough description of the different models in the system analysis activity as well as some guidelines on how to create these models. A recommendation on consistency rules that are relevant for the models in this activity and for the consistency between the models from the previous activity and this activity, is also included.

System Analysis Overview

While the purpose of the requirements analysis is to understand the problem to be solved and the requirements this puts on the system, the purpose of the system analysis is to understand the architecture of the system itself. Essentially the issue of the system analysis is to find out what objects are needed to implement the requirements on the system. This means that the system analysis to a large extent is an analysis of the information that is needed to be represented in the system and the structure of the system itself. Information is here used in a broad sense which includes not only the data to be manipulated in the system but also the containers for algorithms and interfaces.

The system analysis in SOMT is very similar to corresponding activities in other object-oriented analysis methods and the major input and outputs of this activity are illustrated in Figure 645.

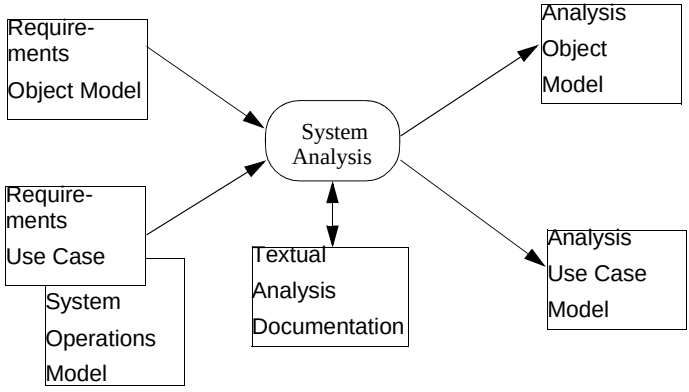


Figure 645: The major inputs and outputs of the system analysis activity

The main input to the system analysis is the requirements object model and use cases developed during the requirements analysis and the main output is another object model, the analysis object model that describes the logical architecture of the system. In addition to this model a use case model is also created in the system analysis to describe the dynamic aspects of the architecture and textual analysis documentation is used to document analysis results not suitable to be express as use cases or object models. The different models will be discussed in detail in the following sections, the analysis object model in "Analysis Object Mod-

el" on page 3863 and the use cases in "Analysis Use Case Model" on page 3875.

The tasks to perform in the system analysis activity are thus essentially the following:

1. Start defining an initial version of the analysis object model, in particular concentrating on creating an overall architecture of the application.
2. Then start refining some of the most important requirements use cases to check that the architecture defined in the analysis object model will work.
3. Continue by iterating between modifying/refining the analysis object model and creating more analysis use cases, either by refining requirements use cases or by describing particular mechanisms in the application.

In parallel with the tasks above it may also be necessary to study various aspects of the chosen architecture, e.g. with respect to non-functional requirements. These results can be documented in the textual analysis documentation.

Analysis Object Model

The intention with the analysis object model is that it is a means to describe the architecture, i.e. the main objects that need to be implemented in the completed system. The notations used are the same as for the requirements object model in the requirements analysis. An overview of the notations is given in "Object Model Notation" on page 3810 and in "State Chart Notation" on page 3814.

At a first glance the requirements object model in the requirements analysis and the analysis object model in the system analysis seem similar but there are several reasons to distinguish between them. The major motivation is that the purpose of the models are different: The purpose of the requirements object model is to investigate and describe the problem that the system is to solve and the environment that the system is to operate in, while the purpose of the system analysis model is to analyze and define the architecture of the system itself. Another more pragmatic difference is that the requirements object model consists of objects visible on the border of the system and outside the system, e.g. users of the

system, while the system analysis object model is focused on the internal object structure of the system.

In the same way as the requirements object model can be structured into a number of different diagrams, the analysis object model can naturally also be decomposed into more than one diagram. This is even more important than it was for the requirements object model since the analysis object model tends to be much larger than the requirements object model.

The Logical Architecture of the System

The major purpose of the analysis object model is to describe an object structure that defines the logical architecture of the application. It describes how the application at a certain level of abstraction can be considered to be divided into a number of subsystems or objects that together fulfil the requirements posed on the application.

For each class that is identified in the logical architecture the most important issue is to note the responsibilities of this class. Why is the class included into the architecture? What is it supposed to do? The responsibilities of an object of a specific class are described by answers to a set of questions:

- What information or knowledge is the object responsible for maintaining?
  - This is described by the attributes of the class.
- What should the object be able to do?
  - Described by the operations of the class.
- What other objects does the object need to know to fulfill its responsibilities?
  - Defines associations and aggregations to other classes.
- Are the responsibilities of objects of this class similar to that of other classes? Is it “the same as class ... except that...”?
  - Defines inheritance relations.

The result when identifying different objects and answering the questions above is one (or more) class diagrams that describe the architecture of the system. The responsibilities are described by the operations

and attributes. The associations between the classes in the model represent needs for objects to be aware of other objects to be able to fulfil its task. As an example, consider Figure 646 that describes the architecture of the access control system.

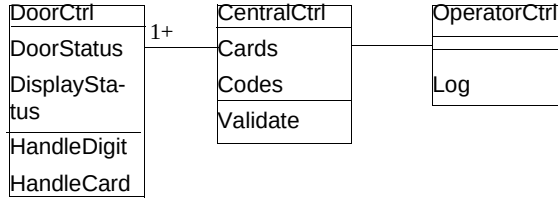


Figure 646: Object model diagram describing the system architecture

Note that object analysis model can be viewed as a refinement of the context diagrams in the requirements object model.

Finding the Objects

The objects in the analysis object model can come from several different sources. Some useful examples are:

- objects from the requirements object model
- objects from interfaces
- objects from use cases

These different sources are described in the following sections.

Requirements Object Model as Source of Objects

The requirements object model is of course one of the major sources of objects for the analysis object model, in particular for the information modeling part. Since the requirements object model should contain most of the objects in the problem domain a lot of them will probably have to be represented in the system and should thus be part of the analysis object model. Note however, that in this case it is not the same object that is found in the requirements object model and the analysis object model. The object in the analysis object model is in this case a container of information about the “real” object that is modeled in the requirements object model. In many cases of course entire inheritance and/or aggregation hierarchies can be reused in the analysis object model as illustrated in Figure 647. Notice that the *Guard* object is not need-

ed to be represented in the system and thus is not introduced in the analysis Object Model.

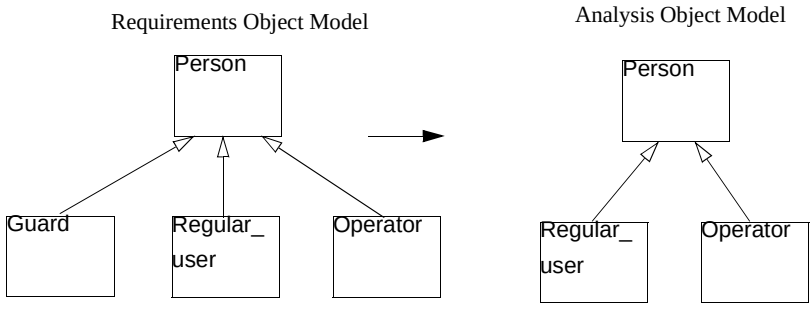


Figure 647: Reuse of requirements object model in the analysis object model

The objects found in the requirements object model usually have one thing in common, they represent entities in the real world that the system needs to store information about.

An algorithm to find the information objects needed by the system based on the requirements object model can thus be phrased as follows.

- For all objects in the requirements object model:
  - Decide if the system needs information about this object to fulfil its task.
  - If the answer is “yes” then add it to the analysis object model using *Paste as* to get the implinks and thus provide traceability back to the requirements object model.

Objects from Interfaces

Another useful way to find objects is to consider the interfaces that the system will have to the environment. It is often very useful to introduce a special kind of object that hides the specific features about how to access the interface from the rest of the system. In [19] these kind of objects are called *interface objects*. Where do we get the interface objects from? There are several different sources to search:

- The application area itself, which in some cases make it obvious what interfaces must exist in the system.

- The use cases from the requirements analysis can be searched for interface objects:
  - They may explicitly identify some interface, e.g. “The user enters a card into the card reader”.
  - Since they define actors that communicate with the system, each actor must use some kind of interface when interacting with the system, even if it is not stated explicitly which interface that is used. So by starting with the actor we may analyze what interfaces he/she will need. If nothing else the actor itself may introduce an interface object.

One important motivation for introducing interface objects is to make modification of the system easier. If the hardware of an interface is changed, e.g. the card readers of an access control system, then the logic of how to handle them is encapsulated in one object. This makes it likely that this object is the only thing that needs to be changed in the software.

Performance requirements are another motivation to introduce interface objects. Very often the interfaces can be a bottleneck with respect to performance. By encapsulating the interface in one object providing high-level operations to the rest of the system it is possible to make an optimized implementation of this object, e.g. making special purpose hardware or enhance the performance.

As an example consider the access control system and specifically the *Enter building* use case described in text in Example 647 on page 3845. From this text we can directly find a number of interfaces: a card reader, a display, a door lock, etc. All of these are likely candidates to result in interface objects in the analysis object model. When describing the interface objects it is usually fruitful to use a communication style class diagram to show how they interact with the external actor and the relevant objects in the analysis object model. If the interaction is non-trivial it might also be a good idea to show the interaction pattern using one or more MSC use cases that describe the different ways this particular object interacts with its environment.

Objects from Use Cases

The use cases can be used as a tool to find the objects that are needed. The strategy is to take a use case and investigate how the functionality that is implied by the use case is distributed among the objects in the analysis object model. One way to do this is to produce an MSC that describes the interaction as described in "Analysis Use Case Model" on page 3875. Check which interface objects are involved, which internal objects are modified or accessed and consider the question of introducing a special controller object to encapsulate the sequencing of the use case. Is there already a control object that might take on the responsibility, or is there a need to create a new control object to handle the logic of the use case?

Consider for example the *Enter building* use case described textually in Example 647 on page 3845 and using an MSC in Figure 638 on page 3847. Since this use case contains a sequence of steps the system needs to represent that essentially has to do with the state of one of the doors and the associated lock and other devices, a special control object *DoorControl* seems natural to introduce. This could give an object model diagram as in Figure 648.

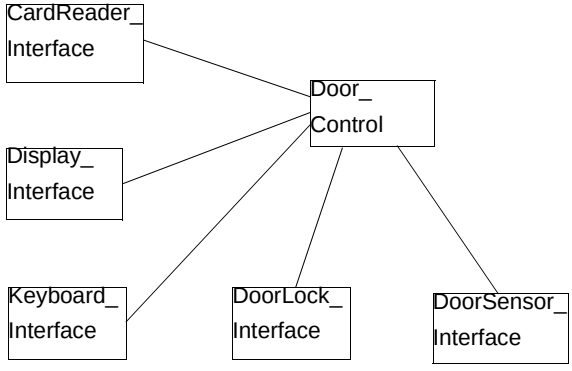


Figure 648: An object model diagram that introduces the DoorControl object

Notice that the objects in this object model correspond to the interface objects that more or less directly could be extracted from the use case text together with the new control object. It may in this context be useful to name the objects according to their function. The interface objects are

in this example called “XXInterface” and the control objects “XXControl” or something similar.

Notice that this model also introduces associations between the interface objects and the control object. These associations represent the communication paths that are needed among the objects. In one way or another information will flow following these associations.

Another type of object that might be found when analyzing the use cases are DataServer objects. These objects define the access possibilities to (complex) data structures. There are several possible sources to search for these type of objects:

- The object structures that come from the requirements object model represent information that has to be stored in the system. In many cases there is a need for a DataServer object that “owns” this information and that provides an access to it. So, investigating the objects from the requirements object model and how they are to be used is a way to identify DataServer objects.
- A second source of information is given by the use cases, since they often express the need for some kind of computation that involves several, related objects. When there is a need for a more complex algorithm that operates on an object structure a DataServer object might be useful.

By once more analyzing the *Enter building* use case described in Example 647 on page 3845 we can see that there is a need for a checking mechanism that determines if the code a user enters is the correct code associated with the card he previously has entered. This indicates the need for a *CardAndCodeDataServer* object that is responsible for maintaining the information about cards and code. Furthermore we can see that a likely operation on this object is a *Validate* operation that tells if one particular combination of card and code is correct. This may give an object model as in Figure 649.

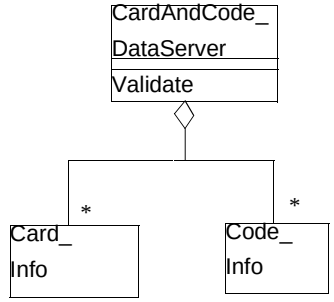


Figure 649: An object model diagram introducing the CardAndCodeDataServer object

Finding Attributes and Operations

As discussed above the attributes and operations are used as a means to describe the responsibilities of objects. They describe the purpose of introducing an object into the model by answering the questions:

- What should the object be able to do?
- What information or knowledge is the object responsible for maintaining?

In practise, the attributes can be found for example:

- In the use cases
- In the requirements object model (by keeping already described attributes)
- In the textual requirements

Some useful sources of operations are:

- The requirements object model (keeping the existing operations)
- The analysis use case model (the messages in the MSC diagrams)

The MSC messages in the analysis use case model can often be considered for operations in the analysis object model. Consider the behavior patterns in the use case model as well, since they often describe the functionality on a more detailed level.

Finding Associations

As discussed above the associations are used to show how object of one class need to know other objects. Usually the associations are found when analyzing the responsibilities of the classes since the motivation for introducing an association is that it is needed by a particular object.

However, some other sources where it is useful to look for the associations are:

- The requirements object model (preserving or modifying existing relations)
- The textual requirements
- The analysis use case model

In particular the last source, the use cases are important. The activity of finding associations in the analysis object model and the activity of constructing the analysis use case model are closely related.

Describing Object Behavior

Sometimes it is useful to describe the behavior of the objects presented in the Logical Architecture. The Analysis Use Case Model describes how objects of the Logical Architecture interact. State Charts describe how these objects reacts internally as a result of the interaction presented in the Analysis Use Case Model.

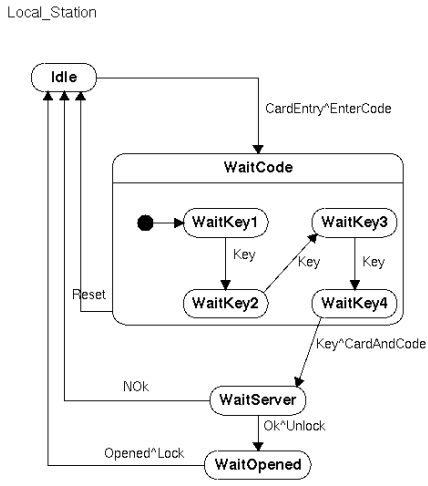


Figure 650: A state chart describing the behavior of an object

It is not necessary to describe the behavior of every object in the Logical Architecture. The focus should be on objects with complex behavior and where complex data structures are dependent of the state of their object. Modeling the behavior of an object will lead to a better knowledge of how the object will function internally. “Which are the actions of the object?”, and “what data structures are needed?” are some questions that may be answered. If an object is too complicated it might be necessary to divide its class into several smaller classes which should be reflected in the corresponding class diagram. In other words: the relationship between the Logical Architecture diagram and the Object Behavior diagram is bidirectional.

When creating a state chart, the Logical structure is the obvious source of information. It is necessary to consider the object’s class specification when specifying its behavior. An operation on a class will lead to transitions in the behavior model. A transition may lead to an internal action or a sending of an event to an external object. A transition may also lead to value changes of the object’s attributes. If these attribute changes are important they should be reflected in the state chart. The operations on the class may also be found in the Analysis Use Case Model together with the possible sending of events.

Architecture of Large Systems

For large applications it is also often necessary to divide the analysis model into more than one module, e.g. to facilitate an analysis by more than one team. One possible strategy for doing this is a recursive approach where we first make an architecture on a high level, where each class represents a subsystem. Then each subsystem is refined by a separate team. It is important to be vary careful about the responsibilities of the classes in the top-level architecture, as described by their attributes and operations, since they will form the input to the different analysis teams.

In this context it is useful to use aggregations to describe a “subsystem” or “is-composed-of” relation, i.e. to describe how the system is decomposed into parts that recursively are decomposed into smaller parts. As an example, consider Figure 651 that describes the structure of the access control system. In this example the system is composed of one or more DoorCtrls, one CentralCtrl and one OperatorCtrl. The OperatorCtrl is itself composed of a component handling remote communication (RemoteCom) and a UserInterface.

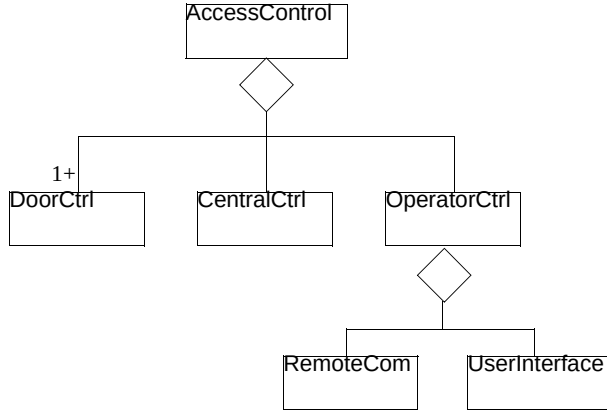


Figure 651: Using aggregations to describe the access control system

Note that the usage of aggregation as a means to describe subsystem relations has an impact on the object structure. Since the assembly classes are used to represent aggregation of classes and not single objects, they should in general not have any “intelligence” by themselves. This implies that the operations provided by the aggregate classes in practise

should be implemented by one or more of the parts classes that the aggregate class contain.

It is a good practise to make the decomposition into modules aligned with the top-level architecture. One top level module architecture of the upper level of the application, identifying the responsibilities of the subsystems. The leaf classes of this top level architecture are then defined in separate modules possible by different analysis teams. The top level module then forms an interface or contract between the analysis teams.

It is impossible to give any rules for how to find the best way to divide a system into subsystems, but most authors agree on some measures to tell if a certain decomposition is a good one. The following two rules are from [5] describing OMT guidelines for decomposition interpreted in the SOMT context:

- The structure must be designed so that most interactions are within the subsystems and not across the boundaries.
- A system should not be divided into too many subsystems, 20 is probably too many. It is better to use a hierarchy of subsystems instead.

The subsystems may be chosen based on several different approaches, but the major idea is to group together objects, that together provide a certain function to the rest of the system, into a subsystem which then can be used as an abstraction of the entire group of objects.

Analysis Use Case Model

The purpose of the analysis use case model is to show the dynamic view of how the functionality of the system is decomposed in the same way as the analysis object model describes the static view of the decomposition. This implies that the internal communication between the various parts of the system is the major concern for this activity. Two different types of use cases can be distinguished in the analysis use case model:

- Refined requirements
- Behavior pattern use cases

Refined Requirements

The analysis use cases that are refined requirements are simply the use cases from the requirements analysis refined to the analysis object model level. Each requirements use case is distributed among the objects from the analysis object model. The purpose of these use cases is mainly to document how the logical architecture as described in the analysis object model is capable of implementing the requirements that are expressed by the use cases. In practice this is done by taking each of the use cases defined in the requirements analysis and reformulate it in terms of the objects that are defined in the analysis object model. It is possible to use both the textual use case notation and the MSC notation to represent the use cases, but since the purpose of the analysis use case model is to show how the functionality is distributed among the objects the MSC notation is especially useful. As in the requirements analysis, HMSCs can be used to simplify the use case model.

Consider again the access control system with a logical architecture according to Figure 646 on page 3865 where the system is divided into a DoorCtrl, a CentralCtrl and an OperatorCtrl components. If we take the *Enter building* use case as defined by an MSC in Figure 638 on page 3847 and replace the system with the three components we get an MSC as shown in Figure 652.

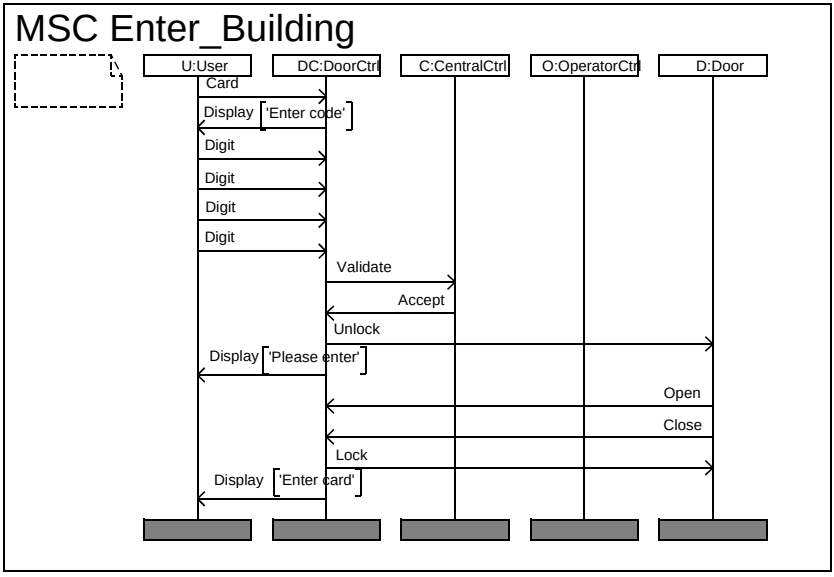


Figure 652: The Enter Building use case distributed over a logical architecture

In this MSC we can see that the original system instance is replaced by instances of DoorCtrl, CentralCtrl and OperatorCtrl. This use case deals mainly with the handling of user interaction at the door and since this is the responsibility of the DoorCtrl most of the action is performed by this component. Only the validation of the card and code is performed centrally.

It is easy to see that the strategy outlined above is extendable to allow a decomposition of a system into not only one level of components, but into a hierarchy of components. For each new level of decomposition all use cases that involve the decomposed components are taken as input to the validation of the new decomposition. The component that was decomposed is replaced by the new components and new versions of the use cases are created.

Behavior Patterns

When designing the object structure for a system there is often a need to document behavior patterns that involve one or more objects that participate to fulfill a common objective. Sometimes this can be described in the refined requirements use cases, but there are two advantages of creating special use cases for specific mechanisms and behavior patterns:

- By describing detailed communication patterns in special use cases, the refined requirements use cases can be on a higher level of abstraction and are not made unnecessary complex.
- By focusing special use cases on specific parts of the system, it is easier to understand and maintain the requirements of the involved object than if these where distributed among all the refined requirements use cases.

As an example consider the keyboard interface of the access control application. The purpose of the keyboard is to allow the user to enter a code consisting of four digits. In a refined requirements use case showing a user entering an office by pushing the digits on a keyboard this can be shown using an MSC reference symbol, as in Figure 653.

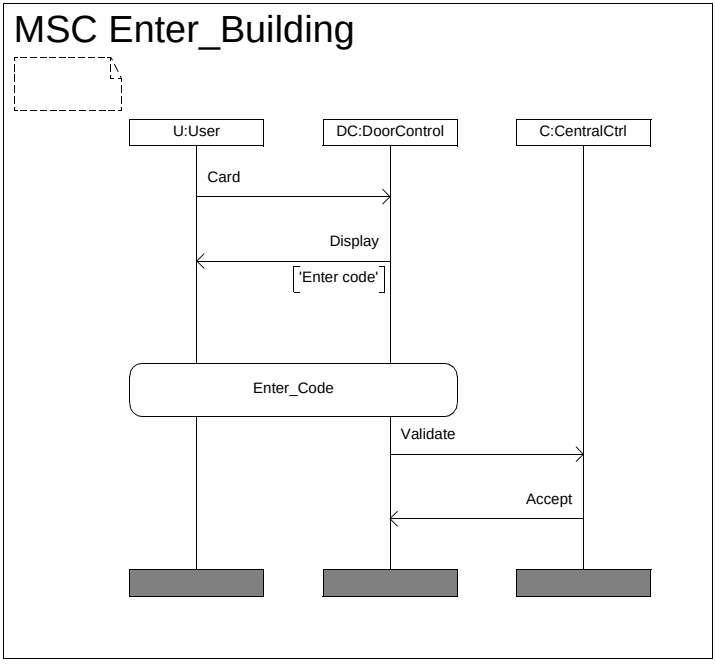


Figure 653: Part of a refined requirements use case

What happens when the user enters a code is thus described in the MSC Enter_Code. We see that it is the user and the DoorControl object that are involved in this interaction.

However, there is of course a specific protocol for Keyboard objects that defines how they interact with the user and the DoorCtrl that is not shown on the abstraction level of the refined requirements use case. This may for example look like in Figure 654, which shows the behavior pattern use case Enter_Code.

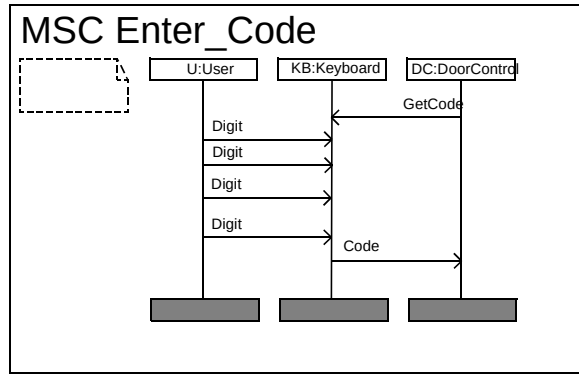


Figure 654: A behavior pattern use case showing the protocol for Keyboard interface objects

Other examples of behavior patterns that may need special use cases are internal communication between different parts of a system using proxies (i.e. local dummy objects that are used instead of a remote object and hides the communication aspect from the user of the proxy), initialization protocols that describe how various objects are created and exchange information, and in general all other tactical solutions to problems that need a special explanation in the system analysis.

Textual Analysis Documentation

In many cases there is a need to express aspects of the architecture or architecture related requirements in a textual format as a complement to the more structured object models and use cases. This may for example be to record experiments performed to check performance aspects of the architecture or other non-functional requirements. Other examples may include documentation of risk assessments performed in the system analysis phase.

To satisfy this need SOMT allows different textual documents to be included among the analysis documentation.

Requirements Traceability

One important aspect in the system analysis is the relation between the models that are created in this activity and the requirements, both external requirements specifications and the models from the requirements analysis. Important questions are:

- Have all requirements been implemented?
- Which are the system analysis objects that implement one particular requirement?
- Which are the requirements that are implemented by one specific object in the analysis object model?

To be able to answer this type of questions it is important to create and maintain descriptions of the dependencies between concepts among the requirements and system analysis concepts. As discussed in "Implinks and the Paste As Concept" on page 3806 the means to do this in SOMT is given by what is called implementation links (or implinks for short). An implink is an association between two concepts where one of the concepts implements the other. One example is the implinks that exist between objects in the requirements object model and the corresponding object in the analysis object model. Consider for example Figure 647 on page 3866 that illustrates how domain objects are modeled in the analysis object model. In this case there should for example be an implink between the *Operator* object in the requirements object model and the *Operator* object in the analysis object model.

Another example is the links between use cases on different levels. Links between the requirements use cases and the analysis use cases show that the requirements are handled on the system analysis level.

In particular when doing "what if..." analysis of the consequences a modification or extension of the system has, these type of links are invaluable. For example consider the case where we would like to specialize the concept of an *Operator* into *RegularOperator* and *ChiefOperator* where the chief operator has some special privileges that a regular operator does not have. We then look at the requirements object model and try to understand what consequences this modification will have. With an implink we immediately see that the analysis object OperatorInfo will have to be changed and with further links from this object we can get a good idea of the consequences caused by the modification.

Consistency Checks

This section provides a list of consistency checks that can be made on the models produced by the system analysis.

- Check that all use cases from the requirements analysis have been refined to analysis use cases.
- Check that all entities in the requirements object model are either represented in the analysis object model or not really needed by the application.
- Check that the object model diagrams and MSCs conform to the static rules for each notation.
- Check that the instances in the MSCs correspond to classes in the object model or to actors that interact with the system.
- Check that the messages received by the instances in the MSCs correspond to operations on the corresponding classes. Note that for remote procedures there may be two messages in an MSC for one operation, one message for the request and one for the reply. In this case the reply message should not have any corresponding operation.

Summary

The system analysis is an activity that is focused on understanding the system to be built. The major tools to facilitate and document the understanding of the system are the analysis object model and the analysis use case model. The analysis object model is intended to capture the objects that are needed in order to describe a solution of the problem.

The analysis use case model describes how the objects in the analysis object model cooperate to fulfill the requirements posed on the system by the use cases from the requirements analysis.

Chapter 72 System Analysis

Chapter

73 *From Analysis to Design*

This chapter describes the differences between analysis and design activities. The aspects to cover when you are moving from object models to SDL, are also described.

From Analysis to Design – Overview

The task of moving from the analysis model of a system to a design of the system is a creative step that involves many design decisions to be taken. The reason is that the purpose of the analysis model is to give an abstract understanding of the system to be built, and an understanding of the concepts that are needed in order to describe a solution of the problem the system is to solve. The purpose of the analysis model is not to give a precise definition of **how** the problem is to be solved or to decide how the architecture and reuse issues are taken care of. These aspects are the purpose of the design activities. Both the system design and the object design are focussed on these type of questions that are necessary to solve in an efficient way in a development project.

The SOMT method introduces a specific concept to emphasize the task of moving from one model to another. It is particularly useful when moving from the analysis model, with its abstract view of the problem, to the design model, with its constructive precise definition of the system that solves the problem. This is the *Paste As* concept that was introduced in "Implinks and the Paste As Concept" on page 3806.

During design the idea is to encapsulate the creative design action taken when moving from the analysis model to the design model into one visible action. From a pragmatic point of view the designer just takes an analysis object and pastes it as a design object. However by doing this simple action the designer documents a number of the design decisions that are involved in the design activity.

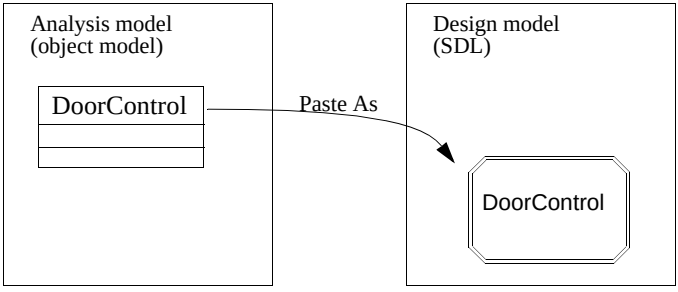


Figure 655: The Paste As concept

One important aspect of the paste-as concept is the relation to traceability links between different models. When performing the paste-as ac-

tion the designer not only creates design objects. He also creates a link between the analysis object and the design objects. This link defines in a precise and compact way the design decisions taken. The links are a vital part of the project documentation that is automatically created by the tools. The links can be inspected when understanding the system and can be used for a number of purposes, e.g.:

- Verification that all analysis concepts are implemented
- "What if" analysis, what happens if one analysis object is changed because of changed requirements

Analysis vs. Design

When comparing analysis to design there is always a question of where the borderline between the activities are. In practise the distinction between analysis and design is somewhat arbitrary. From a practical point of view the important decision is not to define exactly what to call a specific activity, but rather to decide where to document a decision: in the analysis documentation or in the design model. The most important input to guide this decision is to consider the purpose of the different models. The analysis model has a focus on the architecture and most important concepts in the system, abstracting away from implementation details. It is used to provide a means to understand the application as efficiently as possible. The SDL design, on the other hand, is a complete definition of all necessary details.

In the context of mapping object models to SDL in SOMT the trade-off between analysis and design, essentially is a matter of where to document a decision:

- In the object model
- In the SDL model

As an example of this type of question that either can be described in the analysis model or in the design model, consider the issue of the traversal direction of an association, i.e. is the association a one-way or two-way association.

If the choice is to be made in the object model a notation or convention is needed to make it possible to add the extra information. For the one-way/two-way associations a possible notation could be to use an arrow head to denote the direction of the association. Note that this is only an example. SOMT does not recommend this convention.

If the choice is to be documented in the SDL model there is a choice of how the designer expresses his/her choice:, either

- During the Paste-As
- Directly in the SDL model

The result is in both cases the same: The design choice is documented in the combination of the implink (that defines the relation to the analysis model) and in the SDL model (that defines the details of the design choice).

For the one-way/two-way association the Paste-As alternative would imply that the designer interactively entered information as a parameter to the Paste-As of a class to a SDL data type that defined weather an association would be visible in the SDL data type. The association would only be visible if the direction was *from* the given class (or it was a two-way association).

If the one-way/two-way design choice was to be expressed directly in SDL the designer would manually either include or not include the association in the SDL data type corresponding to the class.

SOMT makes a trade-off between the different alternatives to accomplish an as smooth as possible transition from the analysis model to the design model. The SOMT mechanism is based on the following:

- The information that is naturally found in an analysis model is used as far as possible to deduce the mapping to SDL.
- The Paste-As functionality is defined to encapsulate the major design choices, like if an object will be mapped to a process type, struct data type or to some other concept. Paste-As also provides a predefined default choice for all other, minor design choices that have to be made. The intention is that the default settings should be useful in many cases, but in general some of the default choices will have to be modified during the design process.
- The designer will make the final design decisions directly in SDL. This implies that the SDL model is the artifact that documents most of the design choices.

The rationale behind this mechanism is mainly to be able to fulfil the objective of the analysis and design models as good as possible. The analysis model should define the high-level architecture and provide a sound basis for understanding the application as efficiently as possible.

To facilitate this the analysis model should be as abstract as possible while still containing all relevant information. The implication is that design decisions that are not needed to understand the application as far as possible should be kept in the design model and that abstractions should be used as much as possible in the analysis.

Active vs. Passive Objects

One choice to be made when moving from analysis to design is to decide whether an analysis object is to be implemented as an active object or a passive object. An active object is one that has its own thread of control and that can exhibit an autonomous behavior without any other object acting upon it. An active object can be viewed as executing in parallel with other active objects. In a distributed system the active objects may execute on different hardware. In other cases the scheduling mechanisms and operating system creates an abstract interface on which the active objects are executing. One important implication of this is that when two different active objects interact, e.g. when invoking operations, this interaction is a communication between two parallel possibly distributed applications over some communication medium. The communication may be synchronous, e.g. in a client-server relation between objects; the client requests a service from the server and waits for a response from the server before continuing, or asynchronous, e.g. when peer entities are performing a common task using a specific protocol.

A passive object on the other hand does not have its own thread of control. It changes state only when it is being operated upon by another object. Typically a passive object is mainly an information container used in the system to store information about the outside world and the internal state. If the information handled by a system is complex then an entire structure of passive objects may be needed to model it. Often, particularly in distributed applications, there is a relation between the active objects and the passive objects in the sense that an active object "owns" a passive object. The passive object is part of the same thread of control as the active object and it is localized in the same place in the system structure. An example of this might be an active data base server object that handles the control of the access and operations on a data base defined by a number of passive objects. This is very similar to the notion of "container classes" that are used as abstractions of the classes that they contain.

Reuse Issues in the Design Models

Another aspect of objects is if a particular object class is subject to reuse or not. The reuse may be on different levels of ambition. The lowest level of reuse is when an object is reused within one subsystem in an application. This has no major implication on how to package the object definition. Since one subsystem is most often developed by a small group of individuals there are no special requirements on the packaging of the object. The next level is when an object class is found to be useful in different subsystems implemented by different development teams. This situation makes things a bit more complex, since the object definition now probably should be grouped together with other similar objects and put in a special package. This package is used by many of the development groups and there may even be a special team devoted to the development of this support package. In any case it needs its own version control, interface definitions etc.

The third level of reuse is when an object is found so general and useful that it can be used in more than one project for more than one application. In this case there is obviously an even stronger need for a packaging mechanism, since the object together with other related objects now must be seen as a separate product.

Designing an object that is suitable for reuse requires special care: already early in the design it is necessary to identify that an object might be suitable for future reuse. Then the object needs to be specially designed to make it as general (and reusable) as possible. Careful thinking is also required to identify the parts of the object that later might have to be redefined when reused. These parts have to be marked “virtual” in SDL. A rule of thumb is that it is worthwhile to design a reusable object when it is expected that the object can be reused at least three times.

Mapping Object Models to SDL

When moving from an analysis view of an application to a design of the application, one of the major tasks is to define the relation between analysis concepts and design concepts. There are two different ways to view this: either the design is seen as an elaboration and refinement of the analysis model or the design can be seen as a transformation of the analysis model. In practice both viewpoints are useful. When creating the design a transformation is performed from the object model developed in the analysis activity to the SDL design models. The concepts in the object model are mapped to suitable concepts in the SDL domain. However, when this mapping has been done the analysis model is used as an abstract view of the application where only the details relevant from an analysis perspective is present and the design is viewed as an elaboration and refinement of the analysis model.

When mapping object models to SDL there are several aspects to cover:

- Mapping the structure of object models to the structure of SDL
- Mapping the interfaces of objects to interface definitions in SDL
- Mapping the classes in the object model to SDL definitions
- Mapping existing state charts to SDL process definitions

The structure of an object model of interest for the mapping to SDL are of two kinds:

- The aggregation structure in the class diagrams
- The module structure that defines how the complete object model is divided into different modules

The aggregation structure in the object model is in general in SDL represented by the block hierarchy as described in "Architecture Definition" on page 3895. The actual mapping between the object model and SDL involves creating the block structure based on the aggregation structure and is further discussed in "Mapping Aggregations of Active Objects" on page 3917.

The module structure of the analysis model is one of the inputs to the definition of the design module structure. In SDL the major code structuring concept is the package and in general it is recommended to keep a simple mapping from the module structure of the analysis model to the packages in the SDL design. This is further discussed in "Design Module Structure" on page 3897.

In SDL the interfaces of processes, blocks etc. are separated from the description of behavior or implementation of the block/process. This implies that in general there are two different mappings from classes in the object model to SDL, one to the interface definition and one to the process/block definition. Interfaces in SDL are defined using signals or remote procedure calls as described in "Static Interface Definitions Using SDL" on page 3901 and the mapping to these concepts is outlined in "Mapping Object Models to SDL Interface Definitions" on page 3902.

The mapping of the classes in the object model to the corresponding SDL behavior definition is an issue of object design and is depending on if the object is an active or passive object. Active objects are mapped to SDL processes (or process types) and passive objects to data types. These different mappings are discussed in "Mapping Object Models to SDL Design Models" on page 3914.

Summary

When moving from analysis to design, several decisions has to be made. These decisions should be documented and it should be possible to view the impact of such decisions. A way to achieve this is the Paste As functionality that produces links that could be traversed in order to do "what if" analysis or to do consistency checks between requirements and design.

When we start to design from our analysis models, several possible SDL mappings exist for entities in the object model and the use cases. The important decisions that have to be made concern issues like reuse and determining if objects should have autonomous behavior or not (active/passive objects).

Chapter

74 *System Design*

This chapter gives a thorough description of the different models in the system design activity and some guidelines on how to create these models. A recommendation on consistency rules that are relevant for the models in this activity as well as for the consistency between models from previous activities and this activity, is also included.

The chapter requires that you are at least reasonable familiar with SDL.

System Design Overview

One of the most important issues of software development, if not the most important of all, is to define the architecture of the system. Define how the system is built up of smaller parts that in turn may be composed of even smaller parts until each part is manageable by itself. The architecture is proposed in the system analysis architecture but the precise definition of this structure is the major task of the system design activity.

The components of a system may have several different important functions to fulfill:

- They act as a unit for work division. Different development teams can be responsible for different components.
- They form a decomposition of the functionality. Each component may be responsible for one aspect of the total functionality of the system.
- They act as distribution units. The components can define how the system is distributed in the physical world.
- They may act as technology units. The design of the different components may use different notations and tools and, although SOMT has its main focus on SDL, the system design activity also takes other possibilities into account.

The major inputs to the system design activity are the analysis object model and the analysis use case model produced in the system analysis activity. The system design is the process that based on these inputs define in detail how the system is decomposed into components and to define the interfaces between the different parts. This is illustrated in Figure 656 that also shows the three major artifacts developed in the system design; the design module structure, the architecture definition and the design use case model. In addition to these formalized descriptions there is often a need to specify non-functional aspects of the components in a textual design documentation.

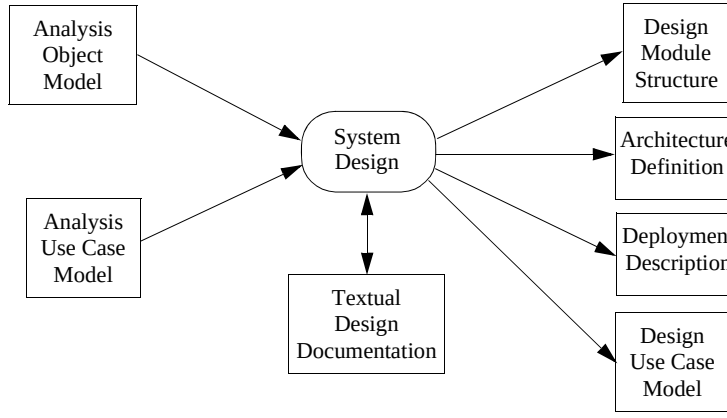


Figure 656: Inputs and outputs of the System Design activity

The architecture is in the system design formalized primarily using SDL. In SDL the major structuring concept is the *block* and the notation is the block diagram. Seen from an object-oriented point of view a block is a container of objects. The block can either be directly described by the objects that it contains or it is decomposed into lower level blocks. The block structuring mechanism is discussed more in "Architecture Definition" on page 3895.

Where the logical architecture defines the decomposition of logical functionality the *design module structure* defines the decomposition into work items. It defines the different modules the design teams can start working on and also provides a mapping from the logical architecture to design modules. In SDL the design modules are usually SDL *packages*. The design module structure also takes a slightly broader perspective of the system to be built and describes how existing frameworks, tools and components are incorporated into the development structure.

The *deployment description* is a way to describe the physical distribution structure of the SDL system. It is also the place where the implementation strategy for different parts of the system can be described.

There should be a simple (if possible one-to-one) mapping between the top levels of the architecture definition and some of the modules in the design module structure. The benefit gained from a simple mapping is

that the design modules define the possibilities to divide the work on different development teams and the logical blocks comprise well-defined sets of functionality and responsibilities. If they do not map to each other there is an obvious risk for complex interfaces between the development teams.

As always when a system is decomposed into smaller parts one very important issue is how the interface between the parts are defined. In particular if the components are used as division of work load and designed by different development teams the interface definition is the means to communicate between the different groups and a common understanding of the interface is crucial. There are two aspects of the interface:

- A static aspect, defining the operations or services that a block offers
- A dynamic aspect, that defines how the different blocks cooperate to solve a common task

Both aspects are important and the definition of them is a vital part of the system design activity.

In SOMT the major concepts used to define the static interface are the SDL concepts *signals* and *remote procedures*, and the dynamic aspect is a continued usage of use cases. However, since there very often is a need to design parts of a system using other techniques than SDL or to use existing modules, other interface definitions techniques are also used in SOMT.

The major tasks to be performed in system design in an SDL based project can thus essentially be summarized as the following:

1. Create an (incomplete) SDL system that is a starting point for the formalization of the architecture of the application. This is further discussed in “Architecture Definition” on page 3895.
2. Define the design module structure. Draw a diagram that illustrates the structure and create the necessary packages etc. as described in “Design Module Structure” on page 3897.
3. Define the physical distribution strategy for the SDL system, see “Deployment Description” on page 3900.
4. Define the static interfaces as discussed further in “Static Interface Definitions Using SDL” on page 3901.
5. Define the dynamic aspects of the interfaces by a continued use of use cases. See section “Design Use Case Model” on page 3904.

There is, as we will see in “Object Design” on page 3911, a close relation between the system design activity and the object design activity in the sense that the object design activity is concerned with the representation and behavior of the objects and the system design deals with the distribution of the objects into blocks and defining the communication paths between the objects.

The rest of this chapter will discuss the system design activity. SDL will frequently be used to define the architecture and interfaces and examples of SDL diagrams will be used throughout the chapter. A complete presentation of the SDL language is however outside the scope of this document. For more information, please consult either the Z.100 standard itself [24], or a text book about SDL like [28].

Architecture Definition

When using SDL to design a system the architecture of the system is defined by the block diagrams. They define how the system is decomposed into blocks and how these blocks either form the leaves of the block hierarchy or are further decomposed into smaller blocks. Essentially this block structure is a formalization of the logical architecture from the system analysis.

As an example, consider once again the access control system. The system controls the doors of a building to unlock the doors when an authorized user wants to enter or exit the building. The task in system design is to define how to structure this system. One natural choice is a distributed structure where the control of each door is localized close to the door and a central controller keeps all common information about authorized users, cards and codes. Furthermore one special block is responsible for the handling of an operator panel. A logical architecture that described this was illustrated in Figure 651 on page 3873 in chapter 72, *System Analysis*. The SDL diagram that shows a beginning of a formalization of this architecture is depicted in Figure 657.

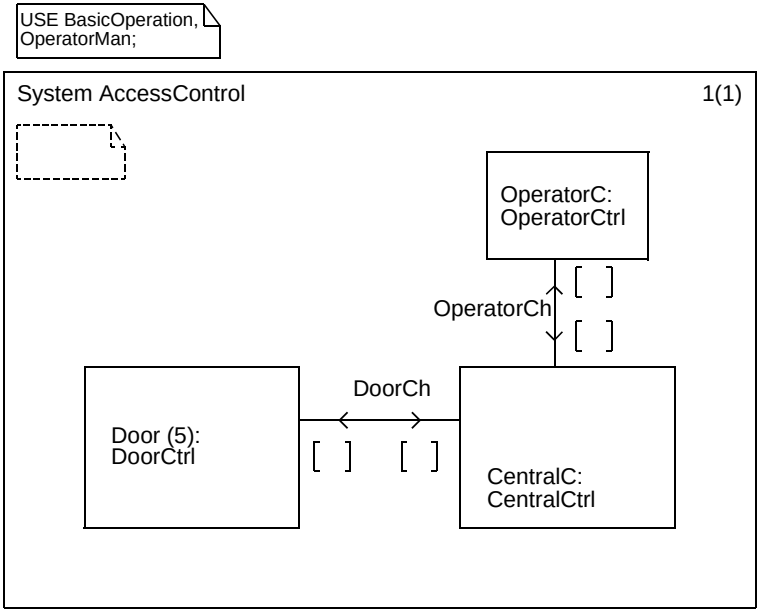


Figure 657: The architecture of the Access Control system defined by an (incomplete) SDL system diagram

In this diagram we can see the blocks *CentralC*, *Door* and *OperatorC* that are instances of the block types *CentralCtrl*, *DoorCtrl* and *OperatorCtrl*. *CentralC* contains the common information base about cards, codes etc. that are registered in the system. *Door* is the block responsible for the control of each door and *OperatorC* handles the operator communication. The diagram also shows how the blocks communicate using the channels *DoorCh* and *OperatorCh*. Note that there are five doors in the building in this case and that this is shown by defining *Door* to be a block instance set. The types *DoorCtrl*, *CentralCtrl* and *OperatorCtrl* are assumed to be defined in the packages *BasicOperation* and *OperatorMan* that are referenced in the *USE* clause in the top of the diagram.

Design Module Structure

The purpose of the design module structure is to show the actual components the application will be built from. The module structure should depict the actual source code modules etc. that the application will contain. A number of different aspects must be taken into account when defining the module structure:

- The implementation strategy for each module: Some modules may be designed in SDL with automatic C code generation. Other modules may be manually designed and implemented in a programming language and yet other modules may need a hardware implementation.
- Existing utility modules that can be reused in the new application
- Existing architectural frameworks that can be reused in the application
- Of-the-shelf utility modules that can be purchased and used in the application

Especially the reuse of existing architectural frameworks is very common and very beneficial. Most applications are not built from scratch, they are rather extensions/modifications of old applications and the design module structure is the place to show how this is done.

One notation that can be used in SOMT to describe the design module structure is object model instance diagrams, where the instances represent the different modules. Where relevant, the attribute field can be used to show what components of the logical architecture are contained in the modules. As an example consider a typical SDL application running on a small microprocessor where a proprietary real-time operating system is used. A possible module structure is shown in Figure 658.

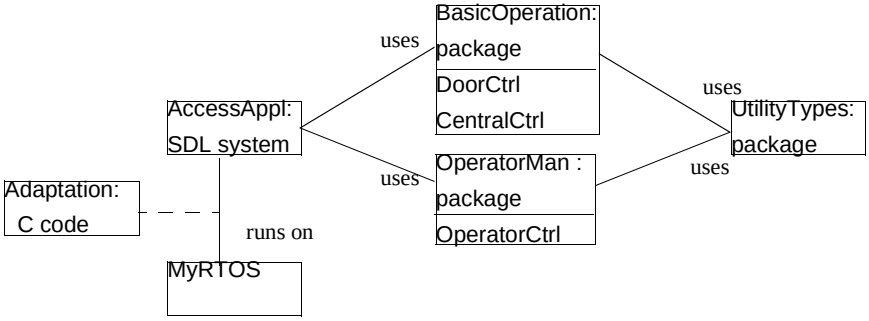


Figure 658: The module structure of the Access Control system using an in-house real time operating system

In this example the top-level of the application is described in the SDL system AccessAppl which is defined using the types defined in the packages BasicOperation and OperatorMan, both of which uses common types defined in the UtilityTypes package. The application will run on the already existing real time operating system MyRTOS. To make the C code generated from the SDL system run on MyRTOS a C code module Adaptation is used that defines the necessary interfaces.

In this example it is very likely that some of the modules will be developed within the project (the AccessAppl and the modules it uses) while others are already existing (like the real time operating system) and some can be taken from previous projects and be modified to fit the current project (like the Adaptation module).

The most important aspect of the module structure is that it forms the basis for dividing the work load on different development teams. This is in many cases the major reason to decompose the system into design modules. However, another reason may be the issue of reuse.

The design for reuse can in this context be viewed as an activity that defines the design module structure based on other premises than the architecture as discussed so far in this chapter. Consider for example the access control system decomposed as in Figure 657 on page 3896, where the system is divided into three different parts according to essentially the physical distribution that is needed in the application. There may in this example exist concepts that can be used in more than one of the subsystems. Examples may include knowledge about some passive

data structure, like the concept of a card, but also entire functionalities like the concept of time seen as a clock functionality.

The identification of these type of components is also a task of the system design. It is particularly important if the different subsystems are to be designed by different design teams. It is also important to identify common components to avoid duplicate work and lower the complexity of the individual subsystems.

The concept that is used to describe the different modules is the *package*. A package is essentially a container of SDL types, this may range from system types and block types over process types down to signals and data types. When a package is used by an SDL system the types defined in the package can be referenced from within the SDL system. For example, in the access control system we may decide that we need a package UtilityTypes (as in Figure 658) that defines the common data types needed in the different subsystems.

Another issue that needs to be handled in the system design is to analyze the consequences of the requirements on different configurations of the system. Are there any optional parts or functionality? In many cases the optional parts are captured by different modules in the system structure, but sometimes, like if there would be an optional requirements on a synchronized clock in all parts of the access control system, it is distributed over the different blocks in the system. If this is the case a package is the most useful concept to use to encapsulate the optional feature.

The discussion so far has been about reuse within one development project focused on one specific application. There is however also the issue of reuse outside the local project. When using an object-oriented approach to the analysis and design the objects tend to be fairly general and applicable in more than one project. If the objects in a particular part of the system are defined as types in a package this will form a good foundation for reusing the objects in future projects. This implies that there may be a reason to use a package structure that is different from the block structure of the system. The package structure reflects the decomposition into packages as defined by the possibility for reuse while the block structure defines the current system structure.

Deployment Description

The purpose of the deployment description is to define the physical distribution of the application and also define the practical details on how to build the different parts.

The specific objectives for a deployment description might vary during different activities: during design we are more concerned on describing on how to verify or validate the design, i.e. how to simulate the design in different ways, while in later activities it is desired to specify the final application build process for the application structure. It is therefor possible to have several deployment descriptions for one system.

There is a textual format for defining a deployment description – for more information, see *"Build Scripts" on page 2642 in chapter 56, The Cadvanced/Cbasic SDL to C Compiler, in the User´s Manual*.

Example of a textual deployment description, i.e. build script:

```
set-kernel SCTAAPPLCLENV
set-env-header on
program UserPart
component system AccessControl / block LocalStation
make-template-file UserMake.tpm
generate-micro-c
program CentralPart
component system AccessControl / block CentralUnit
make-template-file CentralMake.tpm
generate-advanced-c
```

Static Interface Definitions Using SDL

SDL offers two major means to define the interfaces of a block:

- Signals (for asynchronous communication)
- Remote procedure (for synchronous communication)

When signals are used to define the interface to a block they define the communication items that can be sent to and from the block. A signal can represent a service to be carried and it contains all relevant data that is associated with the request. A useful way to structure the signals if one particular interface contains many signals is to define signal lists that group together related signals. Consider the CentralCtrl block above. This block has two interfaces, one to the Door blocks and one to the OperatorCtrl block. The interface to the Door blocks can in SDL be defined as in Figure 659.

```
/* CentralCtrl door interface definition */
signal
  Validate(Card, Code), /* Check card and code authorization */
  Accept,               /* Card and code accepted */
  Reject;               /* Card and code rejected */
signallist CCService = Validate;
signallist CCServiceReply = Accept, Reject;
```

Figure 659: An interface definition using signals

When using signals to define the interface of a block we do not put any constraints on the execution strategies in the respective blocks, we only define the data that is transported. However, in some cases, especially when using a client-server based architecture, it is more convenient to define the interface using remote procedures instead of signals. As an example consider once more the CentralCtrl block. The major responsibility of this block is to store the cards with their associated code. Some possible operations on this data is to check weather a particular card is registered and what the code for a particular card is. A remote procedure definition of these operations is depicted in Figure 660.

```
/* CentralCtrl interface definition */
remote procedure CardRegistered; fpar Card; returns Boolean;
remote procedure GetCode; fpar Card; returns Code;
```

Figure 660: An interface definition using remote procedures

In addition to the signals/remote procedures that are used to define the interfaces in SDL there is of course also a need to define the data types that are visible in the interface. This issue is to a large extent the same as the issue of mapping passive objects to SDL data types. This is treated in more detail in section "Mapping a Passive Object" on page 3921.

Mapping Object Models to SDL Interface Definitions

When mapping object model concepts to SDL there are two aspects that need to be taken care of:

- The design of the interface
- The design of the object itself

In SOMT this implies that an analysis object is seen to have two different descriptions in the design model, one description of the interface and one description of the object itself. In the system design the focus is on the interface definition so we will save the mapping from object models to SDL object definitions until the next chapter ("Object Design" on page 3911). However, defining the relation between the object model concepts and the interfaces between the components of the system is a very relevant issue for this section.

Since the basic mechanism in SOMT to go from analysis to design is using the *Paste-As* mechanism (see "Implinks and the Paste As Concept" on page 3806) this is of course also used when defining the interfaces. As seen in the previous section interface definitions in SDL are defined using signals and/or remote procedure calls. Consequently this is what is produced when mapping a class to an *SDL Interface*.

As an example consider the DisplayInterface objects in Figure 661 that has one operation each.

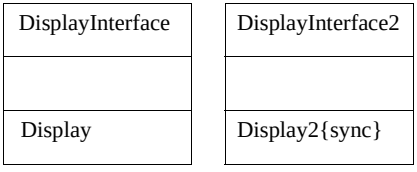


Figure 661: The DisplayInterface objects

When mapping these objects to SDL interface definitions we get in the first case the signal interface definition in Figure 662 (with the signal *Display*) and in the second case the remote procedure definition in Figure 663 (with a definition of the remote procedure *Display2*).

In SDL the natural way is to express interfaces by asynchronous signals, therefore this mapping has been chosen to be the default. (It can also be explicitly denoted with the word *async* within brackets, after the operation name.) If synchronous interface is preferred, this can be denoted by the word *sync*. This is an extension of the original class diagram notation.

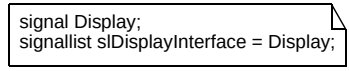


Figure 662: The signal interface from the DisplayInterface object in Figure 661

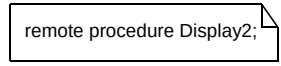


Figure 663: The remote procedure interface given by the DisplayInterface2 object in Figure 661

It is of course also possible to have a mixed signal/remote procedure interface. In this case some of the operations are asynchronous and thus mapped to signals while other are synchronous and thus mapped to remote procedure definitions.

Design Use Case Model

The static interface definition alone is not enough to define how the blocks are supposed to cooperate to meet the requirements on the system. In the requirements and system analysis, use cases were used to describe the requirements on the system. This is continued in the system design to define the dynamic interface between the blocks in the system. Essentially the idea is to take each one of the use cases found in the system analysis and formalize this to a sufficient degree of detail that is consistent with the level of detail that is found in the static interface definitions. The degree of detail must be precise enough to make the design use cases act as detailed test specifications.

A benefit with the design use cases is the structured way in which they are constructed. It is easy to verify that all requirements as expressed by the requirements use cases and refined in the analysis use cases are handled by the design use cases and this gives a formal link between the requirements and the structure of the system that implements it.

In a development environment where the different blocks are developed by different teams they also form a necessary common definition of the responsibilities of their respective blocks and how their blocks are to together fulfill the requirements on the system.

From a practical perspective this puts some requirements on the notation used to describe the design use cases:

- It must be precise and formal enough to allow an specification of test cases on a detailed level of abstraction.
- It should be possible to automatically check the design use cases against the SDL design model.
- There must be a well-defined way to transform the design use cases to executable test programs that can be executed in the target environment against the application.

There are two levels of testing of interest for the design use case model:

- Module testing
- System testing

Module testing is intended to test one specific part of the system and should check that this particular part of the system fulfills its requirements. The system testing is intended to test the integration of the different parts and check that they together fulfil the requirements on the total system.

The design use case models should form the basis for both kinds of testing.

Another aspect of testing is when in the development project it is performed. One of the benefits of SDL is that it is possible to test already on the design model, essentially testing against a simulation of the SDL system. In addition there is of course also a need to test the implementation in the target environment, but if the logics of the application already has been tested during design then the focus of the target testing can be on target integration issues and the risk for logical errors in the design is reasonably small. The design level testing is further discussed in "Design Testing" on page 3950.

In SOMT two different alternative notations are used:

- MSC
- TTCN

Usage of MSC

MSCs can be used in a way that is precise and formal enough and can automatically be checked against SDL design models. A benefit is that it is used also in requirements analysis and system analysis and is intuitive and easy to understand also for non-experts. Notice however that there is a difference between the analysis use cases and the design use cases. The fairly abstract messages exchanged between the instances in the analysis use cases must in the design use cases be refined to the level of the static interface definitions, this may include specifying parameter values that where left out and even replacing one message with a sequence of message exchanges. It may also often be necessary to have more than one design use case for each analysis use case, for example to handle a situation where the analysis use case has left out a parameter and there is a need to test more than one combination of parameters.

Consider again the access control system with a decomposition according to Figure 657 where the system is divided into a DoorCtrl, a CentralCtrl and an OperatorCtrl block. If we take the *Enter building* use case as defined on requirements level by an MSC in Figure 638 on page 3847 and refined to the analysis use case in Figure 645 on page 3862 in chapter 72, *System Analysis*. When further refining this to a design use case we get an MSC as shown in Figure 664 where some of the messages have been refined.

Note that an MSC describes both the requirements on the separate parts of the system and the requirements on the whole system. This implies that the same MSC can be used to define both module and system tests.

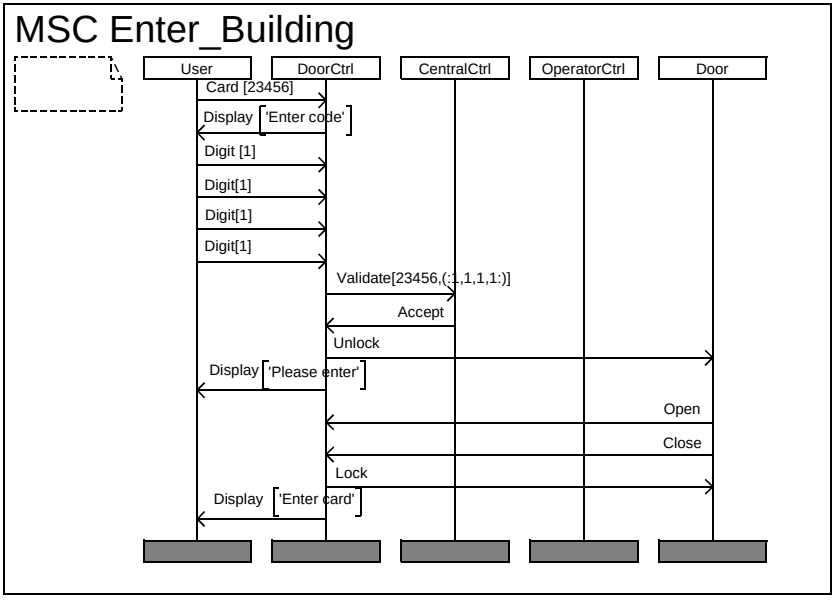


Figure 664: The Enter building use case distributed over an architecture

It is easy to see that the strategy outlined above is extendable to allow a decomposition of a system into not only one level of blocks, but into a hierarchy of blocks. For each new level of decomposition all use cases that involve the decomposed block are taken as input to the validation of the new decomposition. The block that was decomposed is replaced by the new blocks and new versions of the use cases are created.

Usage of TTCN

TTCN is another notation that is suitable for formalizing use cases on the design level. The benefit of TTCN is that it is a special purpose language for test description including:

- Facilities for describing constraints on complex data values
- Preambles and postambles to show how to compose test cases

- Possibilities to handle alternative outcomes of a test case
- A special “verdict” construct to define the outcome of a test case

TTCN is also an established notation for test description so there is good tool support for executing TTCN test cases on target platforms.

The drawback is that it has not a particularly intuitive syntax, making it more difficult for non-experts to maintain, create and review TTCN test suites.

If TTCN is used to define both system and module testing each analysis use case will result in several TTCN test cases, one for each part of the system and then one for the entire system. As an example consider Example 649, that shows a test case testing the requirements from the Enter_Building use case on a DoorCtrl, and Example 650, that shows a test case testing the requirements from the Enter_Building use case on the entire AccessControl system.

Example 649: A TTCN test case testing a DoorCtrl ----------

```
1  UsrPCO?Card                                  Card1
2    UsrPCO?Display                             Enter_Code
3      UsrPCO!Digit                             Digit1
4        UsrPCO!Digit                           Digit1
5          UsrPCO!Digit                         Digit1
6            UsrPCO!Digit                       Digit1
7              CentralPCO!Validate              Validate_1
8                CentralPCO?Accept              AcceptOK
9                  DoorPCO!Unlock               Unlock!
10                   UserPCO!Display            Please_Enter
11                     DoorPCO?Open             Open1
12                       DoorPCO?Close          Close1
13                         DoorPCO!Lock         Lock1
14                           UserPCO!Display    Enter_Card P
```

Example 650: A TTCN test case testing the AccessControl system----------

```
1  UsrPCO?Card                                  Card1
2    UsrPCO?Display                             Enter_Code
3      UsrPCO!Digit                             Digit1
4        UsrPCO!Digit                           Digit1
5          UsrPCO!Digit                         Digit1
6            UsrPCO!Digit                       Digit1
7              DoorPCO!Unlock                   Unlock!
8                UserPCO!Display                Please_Enter
9                  DoorPCO?Open                 Open1
10                   DoorPCO?Close              Close1
11                     DoorPCO!Lock             Lock1
12                       UserPCO!Display        Enter_Card P
```

The choice between MSC and TTCN as design use case notation is very much influenced by application and development organization aspects:

- It is more efficient if the same test definitions can be used both for design level testing and target testing, so the plans for how to perform the target level testing may have an implication for the choice of notation. If the target testing should be done using a TTCN environment then at least the system tests should be defined in TTCN also for the design level testing.
- On the other hand, if the target tests are performed using an in-house test script notation implying that the same notation can not be used both in design and target testing, then MSC has the advantage of being a simpler notation and is already known and used in the previous activities.

Textual Design Documentation

The SDL architecture definition and the design use cases form a specification of the static and dynamic aspect of the components from a functional viewpoint. In many cases there is a need to extend this with more information that is not suitable to express in SDL or as use cases. An example may be a system that requires a user interface with windows, menus etc. or a system with specific requirements on reliability or response times for some or all of the components.

To give the possibility to express this type of specifications and also to allow other types of design or project documentation in an environment that is mainly SOMT and SDL oriented, the SOMT method gives a possibility to include textual documents in the system design documentation.

Consistency Checks

This section gives a number of examples of consistency checks that can be made on the models produced in the system design.

- Check that there is a simple (preferably one-to-one) mapping between all the top-level subsystems in the architecture definition and some of the design modules as defined in the design module structure.
- Check that the actual modules (SDL packages etc.) used in the design are consistent with the design module structure.
- Check that the subsystems in the logical architecture in the analysis object model are mapped to the architecture definition in the design.
- Check that all use cases from the requirements analysis and system analysis are refined to design use cases.
- Check that the instances in the design use cases correspond to the blocks/processes in the architecture definition.
- Check that all objects in the analysis object model either has been mapped to some interface definition or really are internal to their module.
- Check that the different models conform to the rules for their respective notation (like SDL and MSC).

Summary

The system design is an activity in which the architecture of the system to be built is defined in SDL. Use cases from the analysis are refined to a granularity that will be sufficient for describing the behavior of the subsystems in the architecture. These use cases should be a source for module and system testing in later activities.

Chapter 74 System Design

Chapter

75 *Object Design*

This chapter gives you the rules for how to transform object models to SDL design models. Mappings for active and passive objects and associations are described and exemplified. How to define the behavior of an object is also discussed as well as a section on how to test the design.

The chapter requires that you are at least reasonably familiar with SDL.

Object Design Overview

The object design is the activity that completes the definition of the system. It carries on where the system design finished and fills in the details about object behavior and system structure. The object design is a creative activity that refines the object definitions and system structure taking at least three aspects into account:

- The system structure: where in the system is an object localized
- The reuse structure: is the object a subject for reuse internally within the system or externally
- The precise definition of internal object structure, the behavior and data aspects

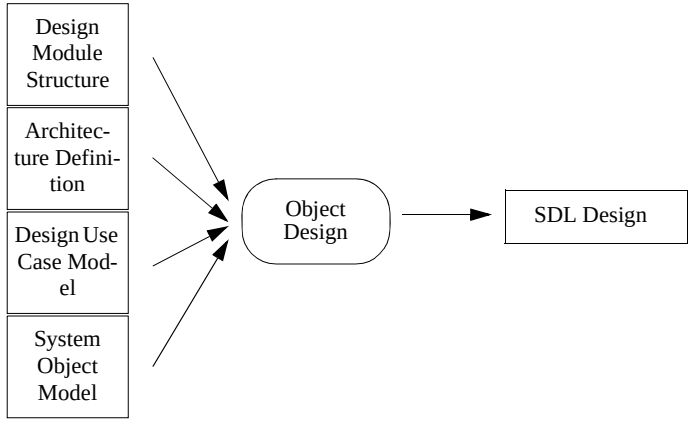


Figure 665: Overview of the object design activity

There is a close relation between the system design and the object design, in the sense that both activities are dealing with the structure of the system. The difference between the activities is mainly a difference in the point of view. The system design takes a top-down look at the system to identify subsystems and overall structures. The object design on the other hand focuses on the objects and uses them as a starting point for the refinement and structuring.

In practice one of the major tasks of the object design is to define the behavior of the objects. Essentially the object design can be viewed as three sequential tasks that must be performed.

1. Map all the classes in the relevant part of the analysis object model to suitable SDL concepts.
   - Active classes are mapped to processes or process types and passive classes are mapped to data types. This mapping is described in "Mapping Object Models to SDL Design Models" on page 3914.
   - The localization of the process (type) or data type in terms of the SDL package/system structure should be done according to the design module structure and architecture definition from the system design.
2. Choose a set of essential use cases and define the behavior of the SDL processes and data types that implements these use cases. Concentrate on the normal behaviors and leave the exceptions for a later step. The design of SDL processes is further discussed in "Describing Object Behavior" on page 3943.
   - Note that this also includes a testing activity that verifies that the SDL design implements the requirements from the use cases. This is further described in "Design Testing" on page 3950.
3. Elaborate the design by introducing more use cases and refine the SDL design to handle also these cases. Take care of exceptional situations like error handling etc.
   - This elaboration is an iterative process where the design is incrementally developed until all requirements are implemented.
   - Each iteration also includes a testing step where both newly implemented and old functionality is checked.
   - The elaboration can include both refining existing processes/ data types and introducing new process/types.

In a project where the design is split on several development teams the iterations in the design will often have to be synchronized. This is further discussed in chapter 77, *SOMT Projects*.

Mapping Object Models to SDL Design Models

In SOMT object design, an important step is the transformation of (parts of) the analysis object model to an SDL system. This section describes this transformation.

There are several aspects important in the transformation:

- How to map objects to SDL, taking into consideration e.g. whether the object is active or passive
- How to represent associations in SDL
- How to preserve the inheritance/aggregation relations in SDL

The mapping to ASN.1 will also be discussed briefly.

Mapping an Active Object

Consider an object in the analysis model that has been found to be an active object. How should this be mapped into the design model? The default choice is to map this object as an SDL process type. The attributes of the object will then be mapped to variables and the operations will correspond either to remote procedures defined in the type or signals handled by the process type. As an example consider the DisplayInterface object in the Access control system. This object is responsible for interfacing a display that is capable of showing a line of text to a user. In the analysis object model this object may look like Figure 666. It has one attribute, *Text*, and one operation, *Display*.

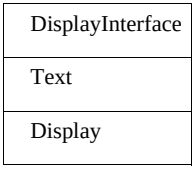


Figure 666: The DisplayInterface object

When mapped to an SDL diagram as a process type the result will be a process type reference, as shown in Figure 667.

Figure 667: The DisplayInterface process type

Now consider the case where the *Display* operation was defined to be an asynchronous operation which is the default. If we now take a look at the DisplayInterface process type definition we can see that it has a variable called *Text* and a gate with the signal *Display*. This process type definition is shown in Figure 668.

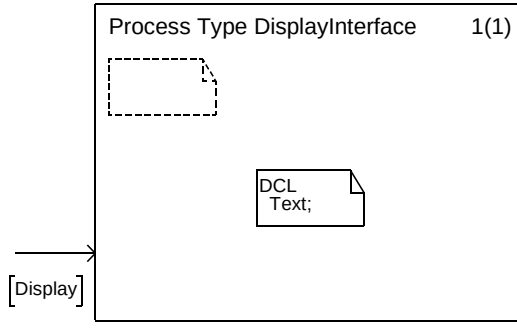


Figure 668: The DisplayInterface process type definition

If, on the other hand, the *Display* operation was defined to be synchronous (by giving the keyword *sync* after the operation name), then the definition of the DisplayInterface process type would now also be different. It would contain a definition of the Display remote procedure instead of a gate with a signal. See Figure 669.

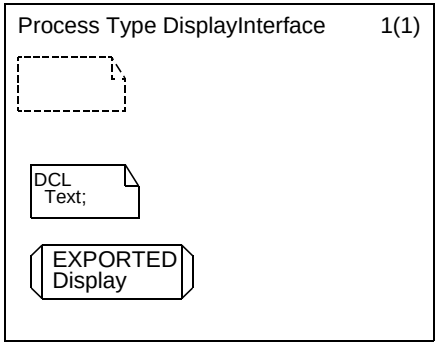


Figure 669: The DisplayInterface process type when mapped to a process type with RPCs

When mapping an object like the DisplayInterface, the design issue of reuse shows up as a question of *where* to define the SDL object. If the DisplayInterface class is considered to be a general class that is to be reused in different projects or by different development teams, the most natural choice is to put the process type in a package that includes this type definition and other related types that are to be reused in different contexts. The design choice of what packages to use in a project was one of the issues of system design.

If the DisplayInterface only is intended to be used in this project but it is used in several of the blocks representing subsystems then there is also a possibility to put the process type on the system level. This would allow the type to be used in the entire system but it would not be convenient to use it in other projects.

If the DisplayInterface is only to be used within the local block it can be defined anywhere in the scope in question, even if it probably is a good idea to keep all process type references on one page in the block diagram.

Finally, if the DisplayInterface object is only to be used in one particular place in the system, it is possible to define it as a process directly where it will be used instead of as a process type. This would be done the same way that it was mapped to a process type except that a process would be generated instead of a process type. Sometimes this strategy is a good idea since it slightly reduces the complexity of the system, however it is only possible if the object is only used in one place in the system.

Mapping Active Objects with Inheritance

Inheritance among active objects is common in object models and is mapped directly to inheritance between the corresponding process types. For example, when mapping the EnglishDisplay object from Figure 670 to a process type, the result is as shown in Figure 671.

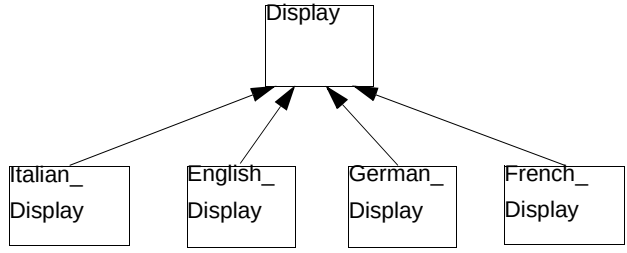


Figure 670: Object model with inheritance

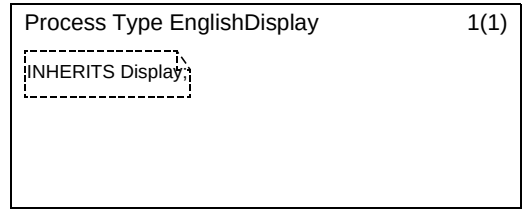


Figure 671: Process type with inheritance

Mapping Aggregations of Active Objects

A fairly common structure is an aggregation with one assembly class that contains a number of other object classes, the part classes[1]. This is common for example when using container classes as abstract interfaces to a certain functionality. One example is given in a description of the software in one part of the access control system: the part that controls the access to one particular door in the building.

1. Terminology (assembly and parts classes) after [21] the “Object Model Notation Basic Concepts”

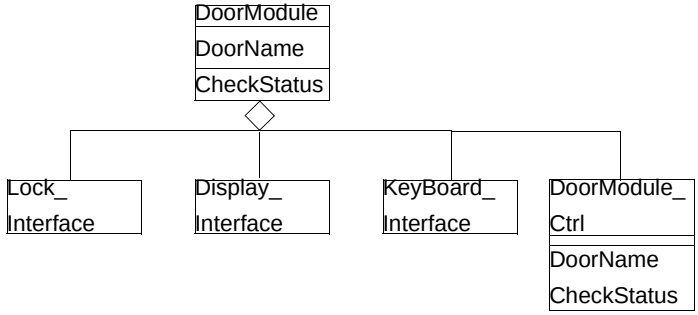


Figure 672: An example of an aggregation of active objects

The most common mapping for this type of structure is to map the assembly class to a block or a block type, depending on the strategy for reuse that has been decided upon for the object.

One special, but fairly common, case is when the assembly object class contains attributes. In this case there are two possibilities: either the aggregation is considered as a “subsystem”, in which case the attributes should be implemented by one of the parts classes, or the object should in addition to the mapping to a block (type) also be mapped to a process within the block that acts as a data server. This mapping follows the same principles as was discussed in section “Mapping an Active Object” on page 3914.

Another possibility is that the assembly object class contains operations. This implies that these operations should be included in the interface of the block/block type. In the same way as when pasting as process types there is a design choice involved here. Are the operations intended to be synchronous remote procedures or are they asynchronous signals? In this case it is also important to define where the operations are to be implemented. Also in this case there are two choices: either the operations are implemented in one of the parts classes or a process is introduced in the block that handles the operations the same way a data server process handles the attributes.

As an example consider the analysis model in Figure 672. The block type DoorModule has both an attribute and an operation. A design counterpart for this is shown in Figure 673 and Figure 674. The process DoorModuleCtrl is the server process that handles the data and operations defined in the DoorModule analysis object.

The choice where to implement the operations and the attributes of the aggregate class is depending on the intended meaning with the aggregation. In SOMT the recommended practise is to use aggregation in object model that describes the architecture of a system as a “subsystem” or “parts vs. whole” relation, meaning the aggregate class is completely determined by its parts classes. In this case the operators and attributes of the aggregate class should be implemented by the parts classes.

Note that this is an example where the attributes and operations of the aggregate class DoorModule have been implemented by one of the part classes, the DoorModuleCtrl class.

Figure 673: A block type reference for the DoorModule object

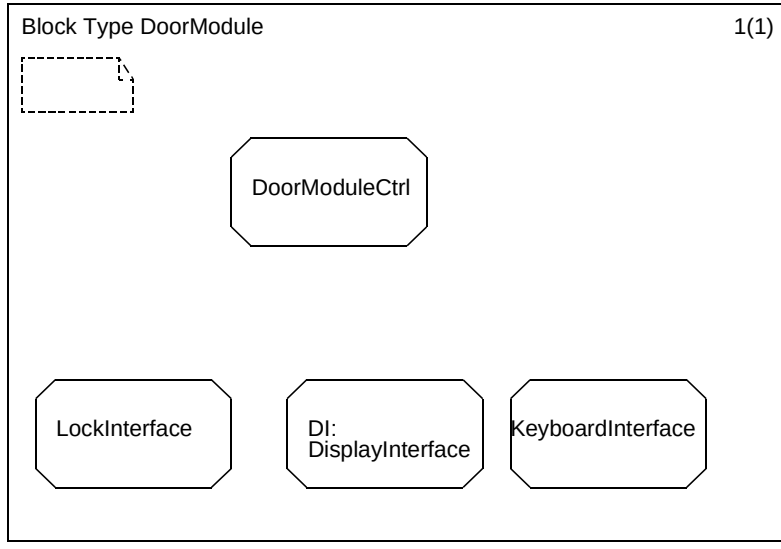


Figure 674: A block type with a data/operations server process

If the object that is mapped to a block type is a subtype of another object in the object model, then the resulting block type will be defined as a

subtype that inherits a block type that corresponds to the supertype in the object model.

Mapping State Charts to SDL Process Graphs

The principle of reusing as much information as possible of the information gained during the analysis activities when working in the design activities implies that an effort should be made to translate state chart descriptions into SDL process graphs. For a thorough description of the mapping rules please refer to: *"Converting State Charts to SDL" on page 1702 in chapter 39, Using Diagram Editors, in the User´s Manual*.

State charts may be converted into SDL. Converting state charts without any hierarchical states is very straight forward, but converting a state chart containing hierarchical states requires flattening since the concept of hierarchical states does not exist in SDL.

The state chart LocalStation Figure 675 contain a hierarchical state with sub states and need to be flattened when converted to SDL.

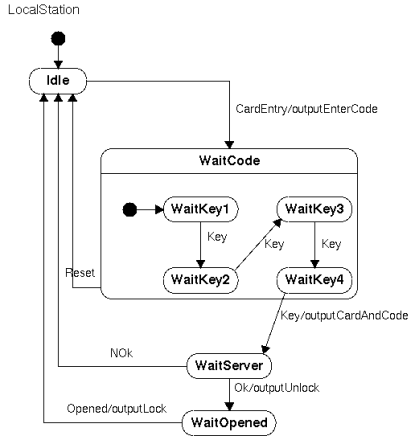


Figure 675: The State Chart Local Station

The behavior of the SDL process LocalStation Figure 676 is the behavior defined in the state chart LocalStation. The process is flattened and gives a straight forward view of the behavior. To increase traceability

between the state chart and the SDL process the converter functionality provides comments on states and transitions which are related to any hierarchical state.

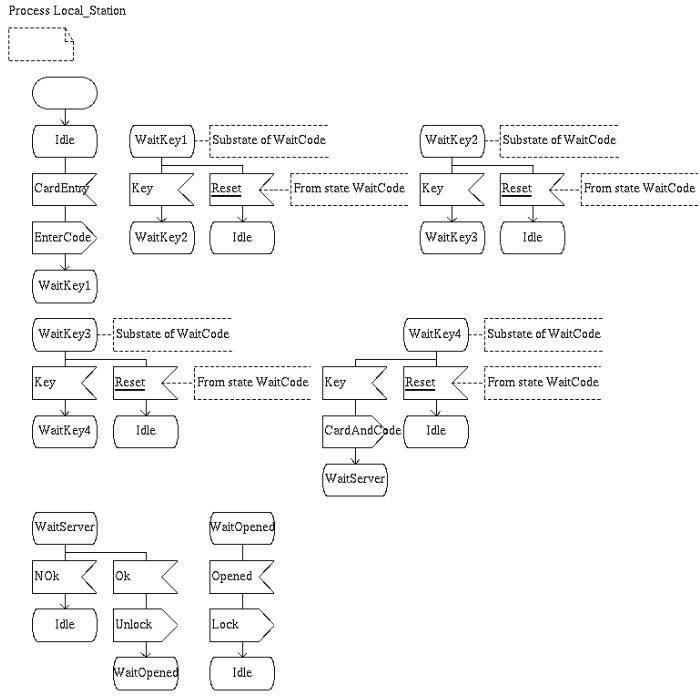


Figure 676: The SDL process LocalStation

Mapping a Passive Object

A passive object is an object that does not have any thread of control or spontaneous behavior of its own. A passive object is often used to encapsulate a certain amount of information that is needed in the system. In the context of distributed systems it is often useful to classify the passive data descriptions into two broad classes depending on the way the data is used: *external* data and *internal* data

- The external data is focused on describing the data units that will be transported across the system when that application executes. In an application where the different components execute in different

memory spaces, and maybe even on different hardware, there is a need to consider transportation format and maybe even coding/decoding of the data. Often the description of external data is focused on the information contents of the data units and not very much on the operations that can be performed on the data units. Typical example applications where external data is very common includes protocols in telecom applications. A common property of this kind of data is that they are usually structured in trees, in an object model often in aggregation hierarchies, and not in graph structures.

- The internal data is characterized by the fact that it is used to describe information that the application needs to do its work. Typically the data is localized to one concurrent execution unit. The internal data units are thus used to store rather than to transport data. Some typical examples are data bases and complex data structures in conventional program units. A local data unit is typically not copied from one component to the other. Instead operators are used to access the data unit.

As will be seen in the next sections the mapping of passive objects to SDL data types is slightly biased towards the external data view of passive objects, the mapping to ASN.1 data types is very biased to external data view, and the mapping to C data types is slightly biased towards the internal data view of passive objects.

Mapping Objects to SDL Structs

In general, the SDL construct that can be used as the design representation for a class is an SDL STRUCT. This is exemplified in Figure 677 below.

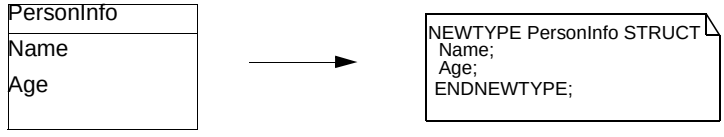


Figure 677: Mapping a passive object to a STRUCT type

In this mapping all attributes of the class are mapped to fields of the STRUCT.

Attribute Data Types

The attributes of a class may have data types associated with them. In general there are three ways to handle these data types:

- Defining the data type in the analysis model.
- Giving an abstract definition in the analysis model and making a final decision in the design model.
- Consider it to be a design issue and give no definition of the data type in the analysis model.

As usual the choice is a matter of personal taste but should be directed by the general idea that the analysis model must be complete enough to be understandable by itself, but as small as possible to facilitate overview and ease of use.

The default mapping in SOMT is a simple literal mapping of whatever exists in the analysis model to the design model in SDL. See Figure 678.

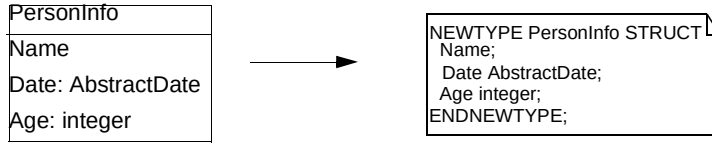


Figure 678: A data type mapping example

A special case which might be of interest in the analysis is when more complex data types like lists or sets are used. SOMT treats this type of data in the same way as elementary types are treated.

Mapping Operations

When a class is mapped to a data type in SDL there exists several ways to map the operations of the class. Operations could be mapped to:

- Operators described by (textual) operator definitions
- Operators described by operator diagrams
- Operators described using C code
- Procedures described in SDL
- External procedures

The default mapping used by SOMT is to map operations to SDL operators with operator diagrams.

Operators Described by Operator Definitions

Operators described by operator definitions is the simplest choice. Consider the PersonInfo class as defined in Figure 679.

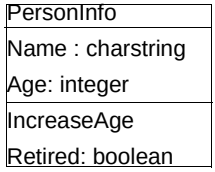


Figure 679: A class with operations

This object has two operations: IncreaseAge and Retired which are intended to increase the age of the person with one, and to check if the person has reached the age where he/she has retired from his job.

Figure 680 shows how a mapping of operations to operator definitions can be done.

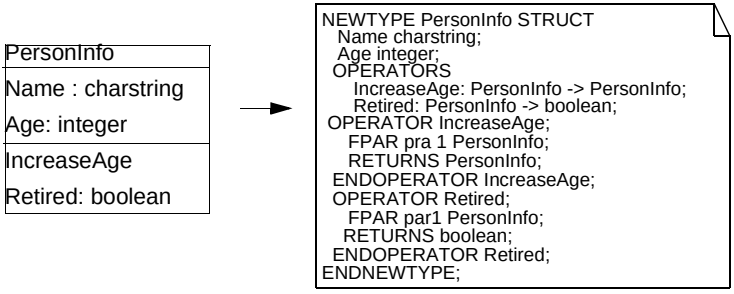


Figure 680: Data type with operator diagrams implementing the class operations

Operators Described by Operator Diagrams

Operators described by operator diagrams is another alternative.

If a mapping to SDL is done using operators with operator diagrams, the SDL will look like in Figure 681.

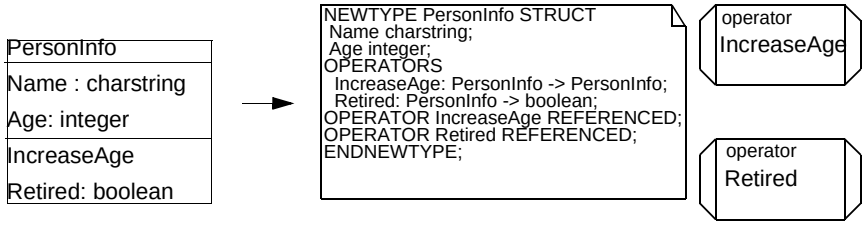


Figure 681: Data type with operator diagrams implementing the class operations

SDL Operators in General

Some things worth noting about SDL operators are:

- An SDL operator can only be used within an expression, e.g. “var:= oper(1) + 1”.
- All operators must return a result.
- Operators can not have IN/OUT parameters. All parameters are IN parameters.

A consequence of this is that an operator cannot both modify an object and return a result. The operators can thus be divided into two classes:

- Modifiers, that modify the object
- Extractors, that extract information from the object but does not modify it

The modifiers are defined as:

```
ModOp: ObjType, P1type, P2type -> ObjType;
```

and used as

```
MyObj := ModOp( MyObj, p1, p2 )
```

An extractor would be defined as:

```
ExtrOp: ObjType, P1type, P2type -> ResultType;
```

and used as

```
Result := ExtrOp( MyObj, p1, p2 )
```

However, SOMT allows a way to overcome the third aspect above. If a C implementation is used as discussed below, the restriction that operators cannot modify the parameters is somewhat relaxed, since the data type itself can be defined using a pointer and then whatever is pointed at can of course be modified by the operator.

Operators Described Using C Code

The second mapping possibility for object model operations is to map them to SDL operators with C implementation. See Figure 682.

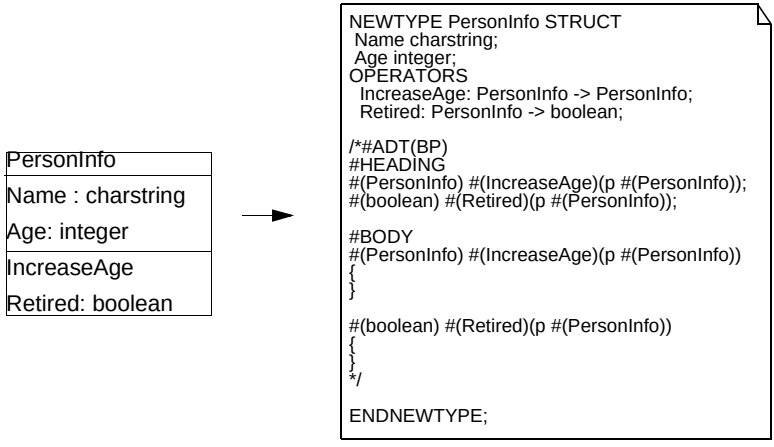


Figure 682: Data type with C implemented operators

Procedures Described in SDL

The third way to implement operations of a class is to use SDL procedures. In this case the definition of the PersonInfo class would be like in Figure 683.

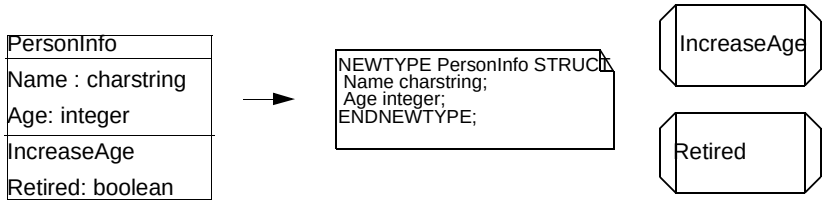


Figure 683: Data type with procedure implementation of operations

Some differences when procedures are used compared to when operators are used:

- Procedures can use both IN parameters, IN/OUT parameters and a return value.
- There is no syntactic relationship between the procedure and the data type definition.
- Procedures can be used in two different ways:
  - using special procedure call symbols (like in Figure 684).

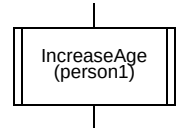


Figure 684: Procedure call symbol

  - in expressions with a syntax as the following example:

```
ret:= CALL retired(p1)
```

External Procedures

The fourth way is to implement the operations as normal C functions. These functions correspond in SDL to *external procedures*. For example, to implement the `IncreaseAge` operation, one could have the following C function (the PersonInfo data type would have to be specified as a C type):

```
void IncreaseAge (PersonInfo *p)
{
  (p->age)++;
}
```

External procedures can be declared in text symbols, and they are called as if they were normal SDL procedures, as shown in Figure 685. "Mapping Passive Objects to C" on page 3935 gives more details about mapping operations to C.

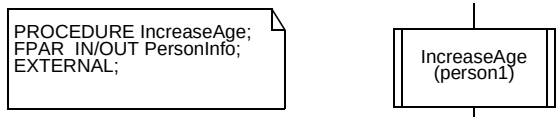


Figure 685: External procedures in SDL

Mapping Aggregations

A common situation is that the information used by a system has to be structured into some kind of tree structure. In the analysis this will appear as an aggregation hierarchy of passive objects. An example is shown in Figure 686.

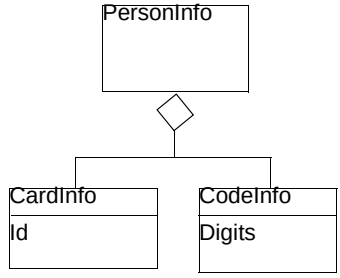


Figure 686: An aggregation of passive objects

If the PersonInfo object is mapped to a struct the aggregation would be visible in the design model as fields in the struct the same way attributes would be mapped. The mapping of the PersonInfo object from Figure 686 into an SDL diagram is shown in Figure 687.

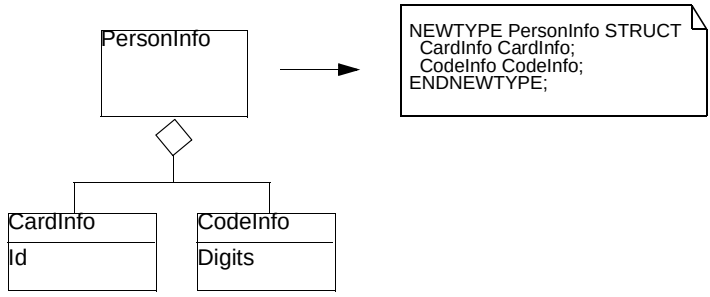


Figure 687: The SDL mapping of a passive object with aggregation

A common special case is when there is a multiplicity associated with the aggregation as in where one person can have more than one card.

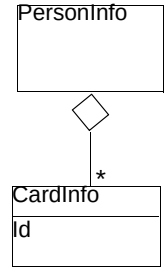


Figure 688: An aggregation with associated multiplicity

In general the multiplicity implies that there is a list or set of elements associated with the aggregation. There are several mappings to SDL possible, for example based on:

- Standard SDL generators like:
  - *Array*
  - *String*
  - Other data types as described in chapter 2, *Data Types*
- A user-defined C implementation of lists

The default mapping in SOMT is a mapping to a type called “xxxList”, where “xxx” is the name of the class. In Figure 689 the mapping of the PersonInfo type is illustrated.

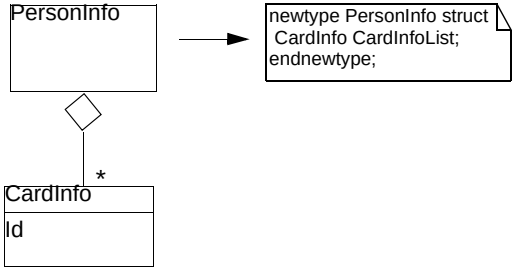


Figure 689: An SDL mapping an aggregation with multiplicity other than one

The “xxxList” type will then have to be designed separately. One simple way is to use an SDL *string* as in Figure 690 for the CardInfoList type.

```
newtype CardInfoList
 string( CardInfo, EmptyCardInfoL )
endnewtype;
```

Figure 690: An SDL implementation of a list using the string generator

Another special case is given by recursive data structures that are used to describe tree structures. In Figure 691 a simple recursive tree is illustrated. In general this type of tree can of course also include a multiplicity other than one.

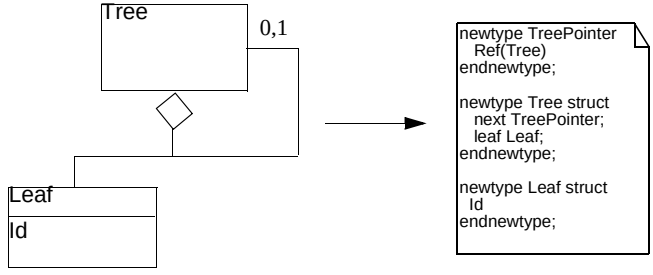


Figure 691: A recursive object model

Due to the lack of pointers in SDL, this type of structure can normally not be defined in standard SDL-92. However, SDL tools may offer tool-specific pointer generators that can be used to represent the above tree. Figure 691 shows how the recursive data structure can be represented with help of the `Ref` pointer generator of the SDL Suite.

Inheritance

Inheritance in an object model is used to model “is-a” relationships. In practise the inheritance shows how attributes, operations and associations are inherited from a superclass to the subclasses. An example is shown in Figure 692 which models the fact that both the users and operators are persons.

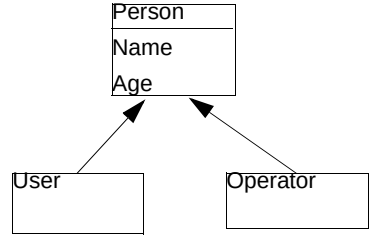


Figure 692: Inheritance relations

When mapping classes that inherit other classes to SDL data types there are three mechanisms that can be used:

- Flattening
- Delegation
- The SDL inheritance concept

Flattening means essentially that all operators, attributes and associations are copied from the superclass to the subclasses. In the example above this strategy would imply that the SDL representation of the User class might look like in Figure 693.

```
newtype User struct
 Name charstring;
 Age integer;
endnewtype;
```

Figure 693: Representing inheritance using flattening

Using delegation to represent inheritance is essentially to replace the inheritance hierarchy with aggregation hierarchies. When using this strategy the SDL representation of the classes in Figure 692 will be as illustrated in Figure 694.

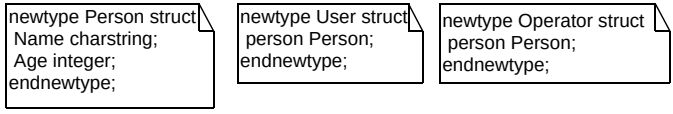


Figure 694: Using delegation to represent inheritance

When using the data types defined as in Figure 694, note that the syntax for accessing the attributes will of course show the delegation strategy used. For example, in order to access the name attribute of a User a construction like “uservar!person!name”” will have to be used.

SDL includes an inheritance concept for inheritance between data types. Unfortunately the inheritance between data types is an inheritance of operators only which limits the usefulness of the SDL inheritance. For more information about inheritance between SDL data types see “Inherits” on page 69 in chapter 2, *Data Types*.

Multiple inheritance implies that attributes, operations and association are inherited from more than one superclass as illustrated in Figure 695, that models the fact that a user is both a person and a card holder.

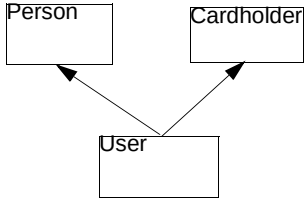


Figure 695: Multiple inheritance example

When mapping classes with multiple inheritance to SDL both the flattening and delegation strategy works fine (except for name clashes when using the flattening strategy) but the SDL inheritance does not include multiple inheritance so it can not be used. In Figure 696 the delegation strategy is used to map the User class from the previous example to SDL.

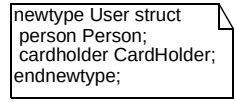


Figure 696: Mapping multiple inheritance using delegation

Delegation is the default mapping of inheritance used in SOMT.

Mapping Passive Objects to Signals

A special kind of passive objects in the analysis object model are the objects that are used only for communication, either between the system and its environment or between modules within the system. For example, when defining the use cases it is useful to show the relations among the events using an object model. It is especially useful if there are inheritance relations among the events as shown in Figure 697. The corresponding SDL signal definitions are shown in Figure 698.

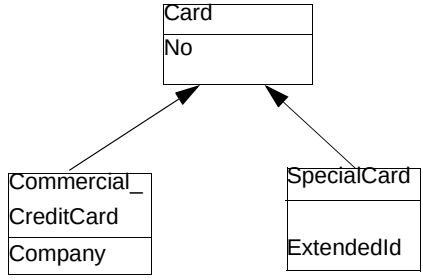


Figure 697: Object model describing communication events

There is often a choice of whether to map a passive object used for communication to a struct data type or to a signal. This is one of the design decisions that has to be taken during the object design.

```
signal Card( No );
signal CommercialCard inherits Card adding (Company);
signal SpecialCard inherits Card adding (ExtendedId);
```

Figure 698: SDL signal definitions corresponding to the object model in Figure 697

Mapping Passive Objects to C

Passive objects can also be mapped to C data types and functions. In C, classes correspond to types, attributes of a class to fields in struct types, and operations to functions.

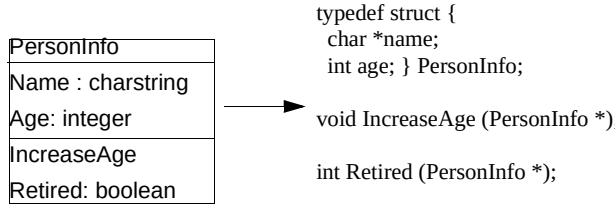


Figure 699 Mapping a passive object to C

Figure 699 shows a possible mapping of class PersonInfo to C. Note that in the mapping of the operations, a parameter is present to identify the PersonInfo object.

To represent aggregations in C, the same mechanisms as described in "Mapping Aggregations" on page 3928 can be used. When mapping inheritance to C, the flattening and delegation mechanisms described in "Inheritance" on page 3931 can be used.

When mapping associations to C, pointers are very powerful. Figure 700 shows an example of a one-to many association that is mapped to a linked list in C. Note how the role name has been used in the mapping.

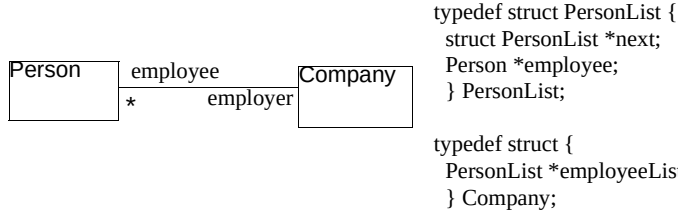


Figure 700 Mapping an association to C

The SDL Suite has facilities to instantiate classes that have been mapped to C, and access their attributes and operations from SDL. This is described more detailed in chapter 2, *Data Types*.

Note:

Try to avoid pointers to data in other SDL processes! Data inconsistency will occur if the same data can be read/written by more than one SDL process at the same time. This can be achieved by avoiding pointers in parameters of signals and remote procedures.

Figure 701 shows an SDL fragment that uses the C types for PersonInfo of Figure 699. The `CString2CStar` operator converts an SDL `Charstring` to C's `char *`. Operations are called by means of SDL procedure calls. Note also the address operator '`&`' in the call to `IncreaseAge`.

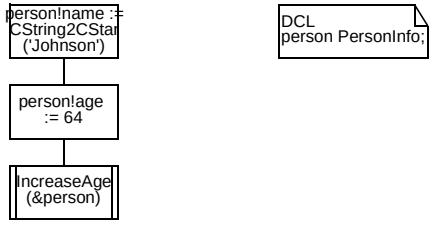


Figure 701 Using C type PersonInfo in SDL

The choice between the SDL struct representation of classes and the C code representation is to some extent a matter of taste. There is a trade-off between simplicity and expressiveness. The SDL struct definition is very simple and all SDL based tools can completely analyze and manipulate this type of data. When using a C implementation, you take over some of the responsibilities from the tools. This means more work for you, but also a possibility to define in detail what you want.

Mapping Passive Objects to ASN.1 Data Types

An analysis object can also be mapped to an ASN.1 data type. The ASN.1 SEQUENCE construct corresponds best to a passive object with attributes. The most basic example is illustrated in Figure 702 below.

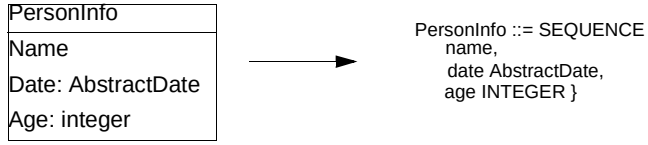


Figure 702: Passive object mapped to ASN.1

The ASN.1 SEQUENCE can be compared with the SDL STRUCT, so most of chapter "Mapping a Passive Object" on page 3921 on mapping classes to STRUCTs is also valid for ASN.1. The few differences are treated here.

The largest difference is that ASN.1 has no possibility to specify operators. Therefore the operations of a passive object should be inserted in some dummy SDL type, while the attributes are mapped to a different ASN.1 type, as illustrated below.

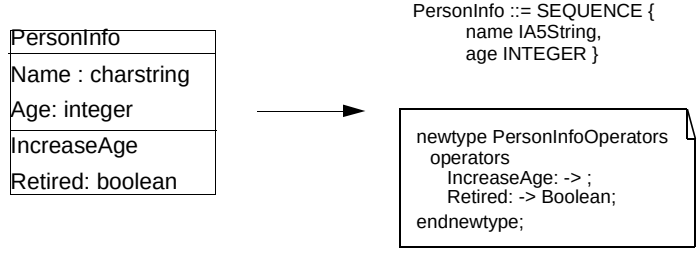


Figure 703: Passive object with operators mapped to ASN.1

The second difference is that ASN.1 has no concepts for inheritance. Therefore only flattening and delegation can be used to represent inheritance. ASN.1 has a special construct, COMPONENTS OF, that is useful to represent flattening. An example of this is given in Figure 704 below.

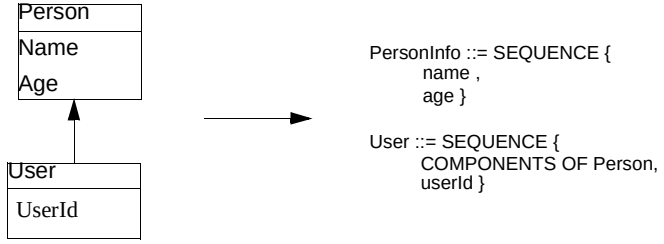


Figure 704: The use of COMPONENTS OF to represent inheritance

The SDL Suite only allows use of ASN.1 in separate ASN.1 modules, that have to be included into SDL with the IMPORTS construct.

Mapping Associations

Associations are in object models used to represent relationships between objects of different classes. The instances of associations are called *links*. An example is given in Figure 705. In this example the association is a relationship between Cards and Codes that describes that a card must have exactly one valid code. In the object model this is represented by an association called Valid between the classes Card and Code.

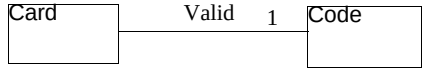


Figure 705: An example of an object model with associations

In the mapping to SDL, the information about the association is kept in one or both of the involved classes, in the example in Card, in Code, or in both.

There are several aspects of associations that are important to consider when mapping an association to SDL:

- Whether involved objects are active or passive
- The structure of the associations: graph vs. tree
- Intrusive vs. non-intrusive representation
- The traversal direction, one-way vs. two-way
- The multiplicity of the association
- Role names vs. association names

- Association attributes/association class

The rest of this section will discuss these issues and how they influence the mapping to SDL.

In general, associations form a graph structure among the objects where each object may have associations to a number of the other objects. Mapping associations to SDL implies in most cases that some kind of object references or pointers must be used.

In general, associations between active objects indicates that there is a possibility for the objects to communicate. In SDL this means that there must exist a communication path (signal routes and channels) between the corresponding SDL concepts (usually processes). In addition, an association between two active objects may also need to be represented by a PId variable in one (or both) of the processes.

There are however cases when PIds should not be used: as PId is a concept that only exists in the SDL world, associations with objects outside the SDL system cannot be represented with PIds. Instead, data types defined in protocols define how to refer to external objects. For example in the TCP/IP protocol, TCP services are addressed by a host address (for example 3.1.29.1) and a local port number. In an SDL implementation of a TCP service it would be impossible to refer to a service outside the SDL system by means of a PId; the host number and the port number must be used instead.

Since strict SDL does not include a pointer concept, associations with passive objects must be represented by other data types. Alternatively a C representation can be used when mapping the objects to SDL as discussed in "Mapping Objects to SDL Structs" on page 3922.

There is one special case where there is no need for special data types. This is the case when the associations form a pure tree structure and furthermore all associations are one-way one-to-one associations with a traversal direction from the root to the leaves. In this case a strategy based on mapping classes to SDL structs can be used, i.e. a reference to an object is in SDL represented by the object itself.

Sometimes an association is implicitly represented in the class definitions themselves, which is called an intrusive representation. In this case the association does not have a representation of its own in the SDL model. For example, if the classes are represented by SDL structs then the association can be represented by a field in one (or both) of the

structs. As an example the Valid association from Figure 705 is mapped in Figure 706 to a field in the struct that represents the Card.

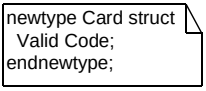


Figure 706: Mapping an association to a field in a struct

A non-intrusive representation of the association, on the other hand, does not rely on fields in structs that represent classes or any similar strategy. Instead the association is explicitly represented in SDL. A convenient way to accomplish this is to use the SDL array generator as exemplified in Figure 707.

Figure 707: A non-intrusive mapping of an association using an SDL array

The design choice to make when choosing a mapping to SDL depends on the way it will be traversed in the application: is the traversal of the association always in one direction or is it traversed in both directions. This is usually not relevant in the analysis model but influences the SDL representation. For example if the Valid association in Figure 705 is a one-way association only traversed from the Card to the Code then the SDL representation in Figure 706 is all that is needed. If, on the other hand, the association is also traversed from the Code to the Card then the Code representation will also have to contain an element corresponding to the association (note that this requires a C implementation due to the lack of pointers in SDL) or alternatively a non-intrusive representation can be used.

The multiplicity of the association defines how many instances of one class can be associated with an instance of the other class. A one-to many association requires a list or set representation. This is essentially the same as an aggregation with a multiplicity greater than one, which is described in "Mapping Aggregations" on page 3928.

Role names are an alternative that can be used instead of or in combination with association names. A role is one end of an association, and the role name uniquely identifies the object from the perspective of the ob-

ject at the other end. An example is given in Figure 708 where the association is identified using role names instead of an association name.

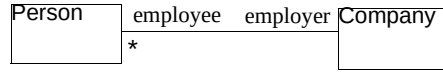


Figure 708: An association that uses role names instead of an association name

In many cases the role names are more convenient and less confusing than an association name. In a mapping to SDL it is preferred to use the role name as e.g. the name of a field in a struct rather than the association name. As an example consider Figure 709 where the Person object from Figure 708 is mapped to a struct in SDL and the role name is used as the field name.

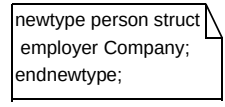


Figure 709: SDL mapping of a an association using the role name instead of the association name

An association may have an associated class as illustrated in Figure 710.

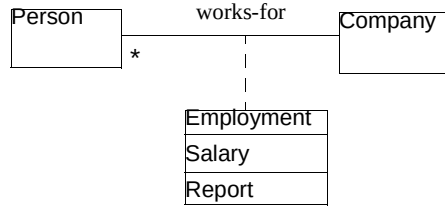


Figure 710: An example of an association class

The implication of this is that there are some attribute values and/or operations associated with each link of this association class that exists between the two objects. When mapping this construct to SDL there are two strategies: an intrusive representation and an explicit representation of the association class. If an intrusive strategy is used there is no explicit representation of the association so the attributes/operations must

instead be associated to one of the objects involved in the relation. This strategy is possible only if at least one of the end points involved has the multiplicity “1”. In the example in Figure 710 it would be possible to incorporate the association class into the Person object since there is only one company for each person in this model. A structured way to represent this in SDL is illustrated in Figure 711.

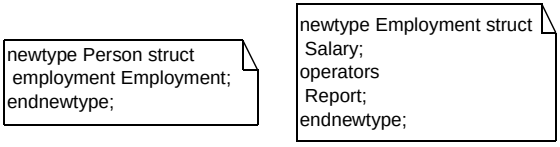


Figure 711: The association class from Figure 710 mapped into the Person object with an intrusive strategy

The second alternative is essentially to view the association class as a regular class and map it to SDL using any of the strategies that can be used to map regular classes to SDL. This is necessary for many-to-many associations.

The default mapping of associations between passive objects in SOMT is to use the intrusive strategy where classes are mapped to SDL structs and thus associations are mapped to fields in these structs. The name of the field in the struct is the corresponding role name if a role name exists, otherwise it is the association name. If there is no name specified for one of the roles and furthermore no name for the association, then no field is generated in the corresponding struct, and the association is considered to be a one-way association.

Summary of Mappings from Object Models to SDL

There are a number of possible SDL target concepts that an analysis object can be mapped to. The choice of target depends mainly on properties of the object:

- Active objects are mapped to processes or process types
- Aggregations of active objects are mapped to blocks or block types
- Passive objects are mapped to struct data types (or to signals if they are used for communication only)

- Associations between passive objects are mapped to fields in struct data types, or to new data types that explicitly represent the association
- Associations between active objects are mapped to communication paths (and possibly to variables within processes)

Describing Object Behavior

Once the type of SDL target concept has been chosen, the behavior of the object can be defined. Processes and process types are defined by creating the process graphs and ADTs are preferably defined by giving operator diagrams for the operators.

In practise, the major task of the object design activity is the definition of the behavior of SDL processes since they are the SDL representation of active objects that tend to have a more complex behavior than the passive objects. SDL process graphs provide a graphical notation for extended finite state machines, i.e. finite state machines with variables. In Figure 712 an example of a small process graph that illustrates some of the constructs possible in an SDL process is shown. More details about SDL and SDL process graphs are provided in section "SDL" on page 3825.

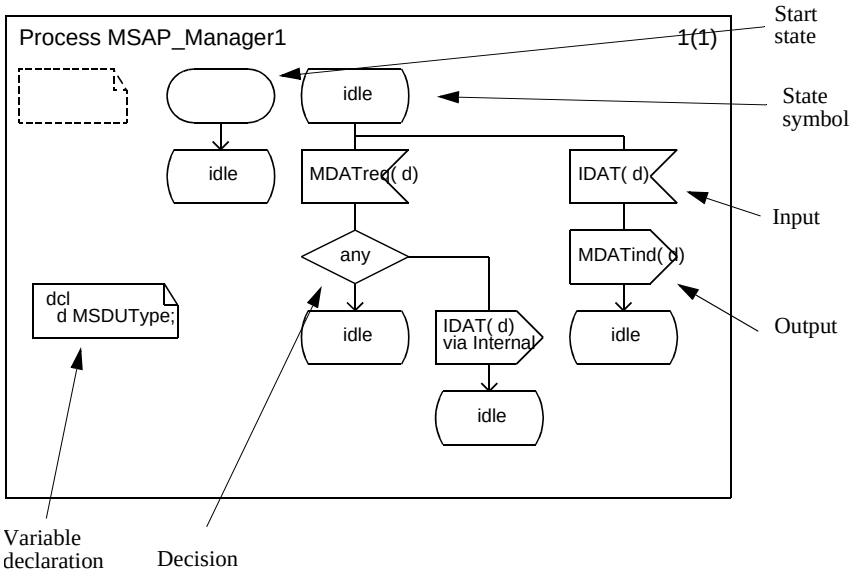


Figure 712: An SDL process graph

The possible inputs to the process design task are illustrated in Figure 713. The possible inputs directly from the system analysis/design are:

- MSC use cases from the design use case model or analysis use case model
- Possibly a state machine, described by a state chart, giving an overview of the behavior if this was defined in the system analysis

The inputs resulting from the mapping of object models to SDL are:

- The signal and/or remote procedure call interface of the process
- Some process variables defined by the attributes of the object model class

Based on these inputs the goal of the process design is to create a process diagram that defines the behavior of the process.

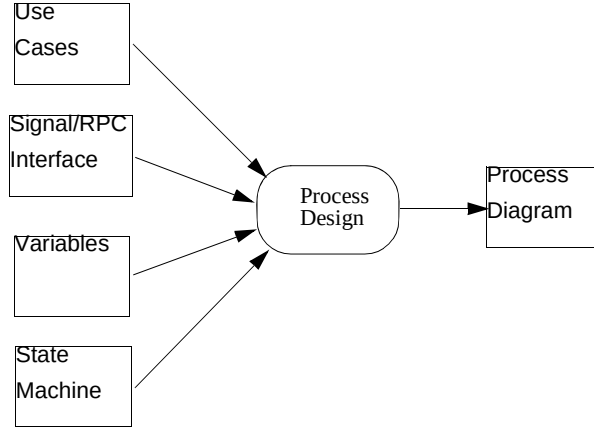


Figure 713: Inputs and output of the process design task

A useful way to structure the tasks that are needed to create a complete process are:

1. Create a first version of the process that defines the control structure in terms of states and transitions. There are two subtasks that in practise are performed more or less in parallel:
   - Defining the control structure
   - Defining data aspects.
2. Elaborate the process by e.g. considering secondary use cases and exceptional cases.

The First Version – Defining the Control Structure

The control structure of an SDL process is defined by the states and the transitions of the process. Ideally the states of a process represent what the environment might expect of the process. Different states represent different stable phases in the life-time of a process and depending on what state a process is in it will respond differently to requests and inputs from the environment.

One straight-forward way to find the states of a process is to analyze the use cases. Since the use cases show an external view of the processes

they reflect the expectations the environment can have on the behavior of the process. Both directly in terms of sequences of inputs and outputs the process has to conform to but also indirectly in terms of states since each input of a process must be preceded by a state.

So, the task of designing the control structure of the process starts with selecting a set of typical and essential use cases. If the use cases are in MSC format and on an appropriate abstraction level that includes the process to be defined, the use cases can be analyzed directly. If not they should be rewritten to clearly show the responsibilities of the process in question.

Analyze each use case in terms of states and transitions for the process. Incrementally build up the process graph by adding states and transitions. Start in the beginning of the use case and figure out what state the process must be in. Manually walk through the use case, checking the process defined so far and incrementally adding states and transitions to the process graph. Check for each transition in the use case (i.e. input followed by one or more outputs by the process in question) that the transition exists in the process graph. If it does not exist, add it. Take a look in the use case to see if there is an external need for a state change in the process: has the expectations on the process changed after this transition in the use case? If no: go back to the same state again. If yes: is there an already defined state that may fit these expectations? If there is one, use it. If there is no such state, create a new one and give it a name that describes the expectations. Continue until all use cases has been analyzed.

When adding transitions do not forget to check with the mapping from the object model if it is a regular transition with input and outputs or a remote procedure that is to be added. Also check if state lists (and “*” states) can be used as the starting state of the transition.

During the design of the process also consider what part of the control to put in the process graph states and what to put in variable values. In general it is recommended to define the control flow using the process graph states, but there are cases when it is better to put parts of the control in data values instead of as explicit process states. One example is loop variables that count the number of occurrences of something and is used to exit the loop after a certain number.

One problem that might occur during the use case analysis is that two of the use cases seem to be very difficult to combine in the same process

since they require different states of the process. This is an indication that some restructuring is needed and that maybe the process should be divided into different processes or into a set of services.

When finished with the analysis of the selected use cases the result should be a skeleton process graph that contains states and transitions with mainly inputs, outputs, remote procedures, timer actions and a few tasks and decisions that deal with control variables. Make sure that the state/transition structure makes sense. The states should represent external expectations on the behavior and their names should reflect it. This is an important issue, in particular for the possibilities to maintain the process.

The next step is now to consider the data aspects of the transitions.

The First Version – Data Aspects

Often there will be three kinds of variables in an SDL process: temporary variables used to handle the parameters of signals, control variables like loop counters as discussed above, and "real" variables that store information about some entity that will be accessed later during the execution. Most of the "real" variables should have been identified during the analysis are given by the mapping from the object model to SDL. The task now is to define how the "real" variables are affected by the transitions. Add temporary variables handling the parameters of the signals when needed and tasks with expressions that define the needed computations. If complex computations are needed it is good practise to hide them in procedures or operators. The transitions should preferably stay fairly simple.

The first version of the process is considered to be finished when it is possible to verify that the process fulfills the selected use cases, e.g. by running a simulator or verifying MSC use cases (compare with "Design Testing" on page 3950). Both the control and the data aspects should be dealt with.

Now it is time to start with the elaboration of the process.

Elaboration of the Process

The purpose of the first version of the process was to define the control structure of the process and make sure that this is able to cope with the requirements from the most typical and important use cases. The pur-

pose of the elaboration is to complete and refine this structure to make the process definition reliable and facilitate the maintenance. There are several aspects to cover in this elaboration:

- Secondary use case and exceptional cases in primary use cases
- Simplification e.g. using state lists and procedures
- Robustness and completeness of the process
- Restructuring for inheritance and reuse

The major topic for the elaboration task is to consider the secondary use case that was not treated in the first version and also the exceptional cases of the already treated use cases. This is done essentially the same way as when creating the first version as described in "The First Version – Defining the Control Structure" on page 3945. The use cases are walked through by hand and the process graph is checked and possibly extended to cope with the new cases.

To enable the understanding of the process and thus also to make it possible to maintain it, it is important that the definition is as simple as possible. This is a topic that is dealt with in the elaboration task. Procedures can be used to simplify process definitions considerably by defining a particular piece of code in one place and using it in several. Procedures can also be used on a higher level to indicate different phases in the lifetime of the process. Using state lists and "*" states it is also possible to simplify the definition of a process by defining transitions that are common to many states in one place.

The robustness and completeness of the process must also be handled in the elaboration. The strategy is essentially to make sure that the appropriate action is taken by the process, not only for the expected cases but also for unexpected cases. So, for each state in the process and each input signal/ remote procedure call possible; check that the action taken makes sense. Also check the treatment of unexpected parameter values.

Another topic to be treated in the elaboration is to consider how to facilitate reuse of the created process. Is it possible to create a more general process type by factoring out some parts of the definition and defining a more general process type that can be specialized in other situations?

The elaboration is in practise an iterative process where all the aspects above are treated more or less in parallel. When the elaboration is fin-

ished, the process definition is completed and is ready for integration test. The module test should preferably already have been done at this stage.

Operator Diagrams

When defining the behavior of passive data types defined in SDL the preferred way to define the operator is using operator diagrams. An operator diagram is essentially a flow graph with a start symbol, symbols defining the actions performed by the operation and one or more return symbols. The symbols may for example be tasks with assignments or decisions. An example is given in Figure 714 that shows the operator diagram for an operator *BirthDay* that increases the age of a person by 1.

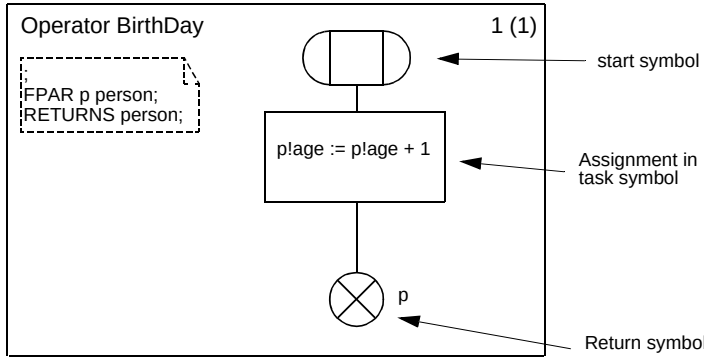


Figure 714: An operator diagram

Design Testing

One of the major benefits with a design notation like SDL that has a well-defined and complete semantics, is the possibility to test the application already in the design activity. This is feasible since the completeness of an SDL design makes it possible to simulate the design taking distribution and concurrence into account.

It is important to emphasis that the output of the object design activity is not only an SDL design but it is a *tested* SDL design that has been shown to fulfil its requirements. This implies that the design testing is an important task in the object design activity.

Testing Strategy

A traditional development/test strategy can be described by a “V” as in Figure 715. The design is performed top down, starting with a system design where the major components and their interfaces are defined, followed by a module design and an implementation phase where the application is implemented. The implementation of each module is then tested separately in a module test and finally the entire system is tested. The system test can also sometimes be divided into two parts, one focusing on the integration of the different modules and the other focusing on testing the functionality of the complete system.

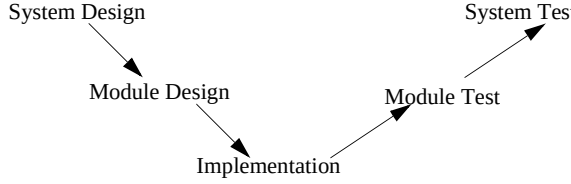


Figure 715: The “V” model of a traditional design/test strategy

This model works in practise fairly good but it has one problem: the complete functionality is not tested until the system tests are performed in the end of the development/testing process. To some extent this problem can be overcome by an iterative process that includes more than one “V” in a development project. Using techniques like SDL this can be even more improved by introducing one more line of testing in the model as in Figure 716.

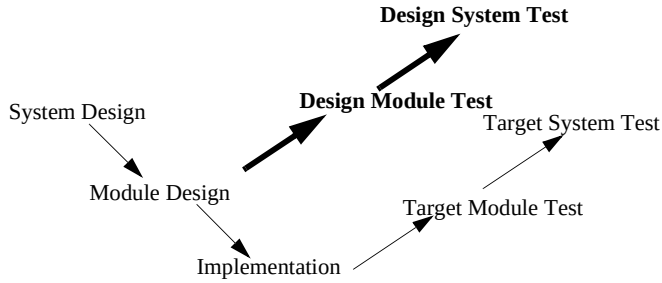


Figure 716: The improved “V” design/test strategy

The extra line of testing represents the design level testing that is the focus of this section. There are two practical aspects that differentiates the design testing from the traditional target testing:

- It is performed using a simulation of the design instead of on the real application in its target environment:
- It is performed as part of the design activity instead of as a separate testing activity performed after the implementation.

The second aspect is very important. Problems that otherwise would not show up until system test time is now found and solved during design at a much lower cost.

In the figure it may look like the testing effort has doubled in the improved “V” strategy since there are now two lines of testing instead of one. This is fortunately not the case. What has happened is that the effort of testing has switched from the target testing to the design testing, making the total amount of testing more or less the same. There are several reasons for this:

- The cost of testing is smaller for design testing than for target testing: the design testing on SDL level is very easy to perform. Since a large portion of the testing is performed at the SDL level this reduces the total testing cost.
- The target testing can focus on the integration and targeting aspects since the logic of the design has already been tested in the design tests. In practice the target module testing can even be skipped if automatic code generation is used when producing the implementation.

Test Case Sources

There are several sources from where the test cases may come:

- From Design Use Cases
- Internally developed during design
- From external sources.

The most important input, both to module and system test, is the design use case model from the system design. This model should capture most of the requirements on the system and using the implinks it is also possible to trace the dependencies from the original requirements all the way to the design use cases. Furthermore, the design use case should be in a format that is possible to test more or less automatically against the SDL system, like MSC or TTCN.

However, during the object design there are usually more test cases developed that tests other aspects of the design, and these form also an important part of the module testing tests.

The third source of tests is external sources. For example, in the telecommunication area it is common to have standardized tests suited for certain types of applications or interfaces. In other cases the customer may have specified acceptance tests that the system must comply to before it is approved. These type of tests should of course also be part of the design tests, in particular for system testing the external sources are important.

It is convenient, but not necessary, if the same notation is used for the design use cases, the tests developed during object design and the external test suites.

Tools for Testing

To perform the testing there is a need for tools, and fortunately there are several tools available that make a number of testing strategies possible:

- Manual or batch simulation of an SDL system using a standard SDL simulator
- MSC verification and automatic testing using a state space exploration tool
- Integrated-simulation of SDL and TTCN, using standard SDL and TTCN simulators

Using a standard SDL simulation tool it is of course possible to manually simulate the test cases and check that the system performs as expected, but it requires quite a lot of manual work. A better way is to produce test scripts that contain the simulator commands that are needed and then execute them automatically in a batch mode and log the results on a log file. To check the outcome of the test either the log files are manually inspected or checked by a post processing tool that e.g. compares the new log files with old, manually inspected log files.

Another approach to testing is to use a state space exploration tool that can automatically check if an MSC is consistent with the SDL system. The benefit with this method is that MSCs can be directly input to the tool and checked and furthermore there is no need for a manual inspection of the results: a verdict can be automatically generated by the tool. The drawback is that some features like the combination of user-written and automatically generated code can not always be handled by state space exploration tools.

A third approach that is useful if the tests are defined using TTCN is to use an integrated-simulation of an SDL system and a TTCN test suite as the means to perform design testing. Essentially this is similar to using an SDL simulator alone, but instead of specifying the input as simulator commands a TTCN simulator specifies the input to the SDL simulator and checks the outcome of the test.

Test Practices

A common situation in particular when performing module tests is that there is a need to test one part of an SDL system in isolation from the rest of the system. In an SDL system the part may be for example a block, a process or a data type definition. The simplest way to accomplish this in SDL is to use a package as a container of e.g. the block (which in this case will have be a block type) or data type and then use a special test system to specify the test environment.

As an example consider the DoorCtrl part of the access control system. Assume that this part is designed as a block type “DoorCtrlT” in a package DoorCtrlPack. To perform a module test on this block the simplest way is to create a special test system DoorCtrlTest that instantiates the DoorCtrlT and connects all its gates to the environment.

Consistency Checks

This sections describes some consistency checks that are useful to perform on the models produced in the object design.

- Check that all objects from the analysis object model has been implemented in the design.
- Check that the design model is complete according to the SDL rules, e.g. that all processes have a defined behavior.
- Check that the design model correctly implements the requirements from the design use cases using design level testing.

Summary

The object design activity should produce a complete and tested design of the system. The precise system and internal object structure as well as the reuse structure are defined in this activity. Relevant parts of the object models are mapped to SDL concept and then completed in an SDL process design activity with design use cases as the main input. This is done iteratively by starting with a initial set of essential use cases, making that part of the design complete and then testing the design (verifying it against the use cases). The activity is finished when all use cases have been implemented and the final design has been tested against all the use cases.

Chapter

76 *SOMT Implementation*

This chapter describes the different ways to implement an SDL design and the tasks that have to be done in connection with this. The focus is on automatic implementation relying on automatic code generation and using C as the target language.

Implementation

The implementation activity results in a tested and running application in the target environment.

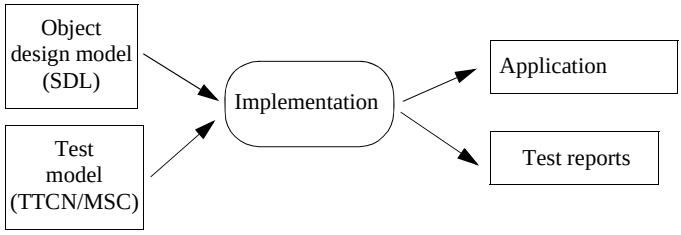


Figure 717: Overview of the implementation activity

The activities during implementation are highly dependent upon the application and target and there are several possible ways to implement an SDL design. It can be manually implemented in software or hardware or a more automatic implementation can be used relying on automatic code generation. The discussion in this section is mainly intended to give an overview of different aspects on the latter approach using C as the target language. There are several tasks to be done in the implementation activity:

- Partitioning the SDL system into different software (and/or hardware) run-time modules.
- Implementing the adaptation code that is needed for this specific SDL system to operate in its environment.
- Selecting and implementing a general integration strategy for the target hardware and run-time environment.
- The generation of C code and the customizing of it for different application areas.
- Testing to ensure that the application works in the target environment.

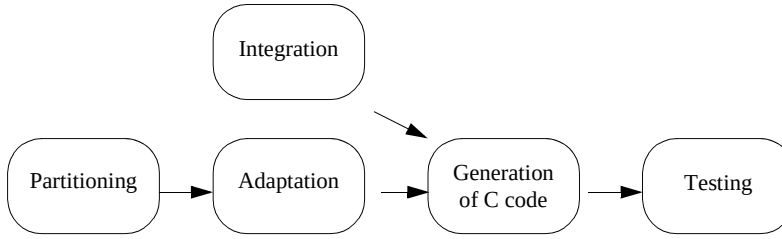


Figure 718: Example of activities during the implementation

Partitioning an SDL System

The purpose of the partitioning task is to partition the SDL system into separate parts that will form stand-alone executables by themselves, either to run as separate programs on one computer or distributed on several computers in a network.

There are two different possible strategies for how to do the partitioning in practise:

- Use directives to the C code generator to define what parts of the system that will form separate run-time executables.
- Create different SDL systems for the different partitions and generate code for them separately.

Adaptation

An application generated from an SDL description can be viewed as having three parts:

- The SDL system
- The physical environment of the system
- The environment functions, where the SDL system is connected with the environment

When adapting the application, the environment functions may have to be specified depending on the integration mechanism used. The environment functions are the place where the two worlds, the SDL system and the physical environment, meet. Signals sent from the SDL system

to the environment can be specified to perform any event in the physical environment, and events in the environment are specified to cause signals to be sent into the SDL system.

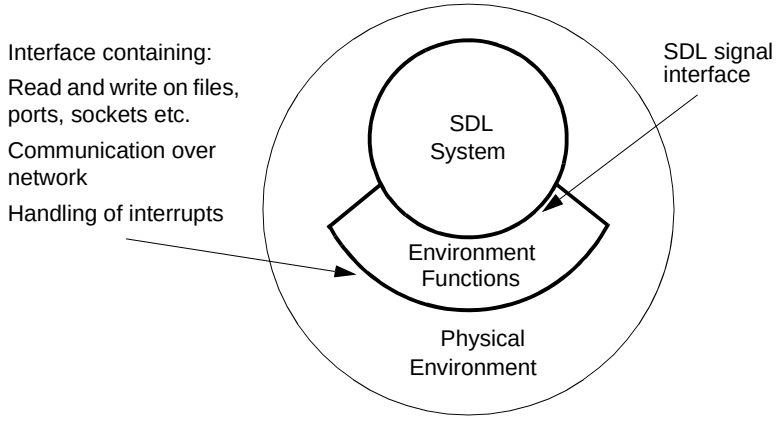


Figure 719: The structure of an application

In a distributed system an application might consist of several communicating SDL systems. Each SDL system will become one executable program. The environment functions that has to be written for each SDL system are:

- `xInitEnv` – handles the initialization of the environment and will be called during the start-up of the application (before the SDL system is initialized).
- `xCloseEnv` – is called when the SDL system terminates.
- `xOutEnv` – will be called each time a signal is sent out of the SDL system.
- `xInEnv` – This function is repeatedly called during the execution of the system. During this call, the environment could be scanned for events which should result in a signal sending into the SDL system.
- `xGlobalNodeNumber` – returns a number that is unique for each communicating SDL system that constitutes an application.

Integration

The purpose of the integration task is to select (and implement if no predefined integration exists) a generic integration mechanism that makes it possible to execute a generated application in a target hardware/software environment. There are three different strategies available for this integration:

- Making an integration using the SDL run-time system, where the SDL application directly runs on a micro processor without any additional operating system support.
- Making a light integration to an operating system, where an SDL system (together with a run-time system) is treated as one task in the operating system.
- Making a tight integration to a real-time operating system, where each SDL process instance (or set of instances of a specific type) results in a task in the operating system.

Both the integration using the SDL run-time system and the light integration use a supplied run-time system to execute the SDL system that takes care of the SDL semantics including scheduling of processes etc.

A tight OS integration consists of a set of files that define how the SDL semantics is mapped to the operating system in question. The main categories are:

- Macro definitions that define the macros in the generated code
- Functions that handle constructs that cannot be used directly in the operating system (i.e. saving of signals)
- Functions that are dependent on the specific operating system (i.e. allocate/free memory)
- Definitions and handling of SDL predefined data types
- Identifiers for signals, timers and remote procedure calls
- Post processing utilities to enumerate signal and process types

An existing integration for an operating system is highly reusable and several predefined integrations are available.

C Code Generation

Although the C code generation is a highly automated activity, there are several ways of customizing the generated code. Some examples are:

- Generating separate C files for different SDL structural entities
- Assigning priorities to processes to enable scheduling (which is undefined in SDL)
- Making a user-defined implementation for the handling of certain signals
- Making a user-defined main-loop (run-time system) in the generated application

Separation of the C files is recommended for large systems, since a minor, local change in for example a block diagram only requires a regeneration and recompilation of the code for that unit. The object files (the compiled versions of the C files) for the other unchanged units can then be used in the link operation to form a new executable program. Thus, the turn-around time from a change in the design to an executable application is minimized.

When doing a customization of the code, it is important that this is done in a way that keeps maintenance of the design easy.

Testing

The purpose of the testing task in the implementation activity is to verify that the application works in its target environment. The input is the test cases specified in the system design and the output is a test report and a tested application. The details of the testing task is very much depending on the target environment and the possibilities to run tests against the application in this environment. Essentially the tasks involved in the testing are:

- Creating a test environment that can communicate with the application and where the tests can be executed
- Transforming the test cases to a format that can be used in the test environment
- Run the tests
- Analyze the results of the tests

Summary

The final SOMT activity, the implementation, completes the development activities by creating an application that runs on the target environment. From the SDL object design, C code is created, the code is adapted to handle the environment of the SDL system and then integrated to the hardware by means of generic interfaces to real-time operating systems. Finally, the application is tested in its target environment.

Chapter 76 SOMT Implementation

Chapter

77 *SOMT Projects*

This chapter describes how to organize the activities and models of SOMT into a project. The description is intended to be suitable for a medium size project.

SOMT Projects

A development project that uses an object oriented approach like the SOMT method is not very different from any other development project. The project has a starting point, the project members work hard and finally the project is finished, hopefully successfully accomplishing the goals of the project in terms of quality, functionality and delivery time. The skill of planning, managing and executing a successful development effort has been studied and described in many books and it is outside of the scope of this document to give a complete treatment of this topic. A recent book that treats management and planning issues for object oriented projects is “Object Solutions” [33] which is recommended for further reading.

Nevertheless, this section is intended to give some indications on how to organize the activities and models of the SOMT method into a project. The description is intended to be suitable for a medium sized project where the effort is big enough to force the development to be divided into several development teams. A smaller project would most likely need less formalism in the development process and a bigger one more control.

Project Phases

From an external point of view (like a manager’s point of view) it is useful to organize a project into a number of *phases*, where each phase represents work performed during a limited period of time resulting in a set of artifacts, deliverables that summarizes the state of the work after the phase is finished. There is a fundamental difference between the phases of a project and the activities as used in this document to describe the SOMT method:

- The phases are used to organize the calendar time of a project.
- The activities represent a categorization of different kinds of tasks by their nature, not by their relative time in a project.

Nevertheless there is a strong relation between the phases of a project and the activities that has to be performed. There is a dependency between the activities of the SOMT method that indicates an ordering in time between the activities. There is no reason to start the system/object design before a substantial amount of work has been done on the system analysis. Likewise there is no reason to start the system analysis before

there is a common understanding of the requirements elaborated in the requirements analysis. On the other hand it would be a waste of time and even dangerous for the project to get bogged down with the details of the requirements for a substantial amount of time without considering their consequences in terms of architecture and design.

The dependency between the activities indicates that a fruitful way to organize a SOMT project is to consider it as a sequence of iterations of the activities. The phases are introduced as a means to get a visibility of important milestones and deliverables from the project. One way to define the phases of a SOMT project is as follows:

- Prestudy/conceptualization phase
- First iteration
  - Requirements analysis phase
  - System analysis phase
  - System design phase
  - Design/implementation phase
- Elaboration phase
  - Planned consecutive iterations

Each of these phases has a section of its own in this chapter.

Multi-User Support

There is often a need to facilitate multi-user support when working with an SDL system managed by SDL Suite. That is, we want to have a development environment capable of handling parallel development. One way to get support for the development process is through using Configuration Management (CM). With CM is usually meant the check in and check out control of sources and binaries, and the ability to perform builds of these entities.

If CM is to be used within a project it has to be planned for. This planning is not something that is carried out in isolation. It is a part of the overall project planning process and is implemented throughout the life cycle of the project. The plan should cover activities to be performed, the schedule for the activities, the assigned responsibilities and the resources needed (including staff, tools and computer facilities). A key

task is to decide exactly which items that are to be controlled and define the engineers responsible for these items. During the development process the code is usually stored in a database based on a revision control system. The developers checkout a copy of it to their own personal workspace. Changes are then made locally and checked in when completed and tested. The developer is responsible for making sure that the module works before he checks in the changed files.

When several users work on an SDL system, the management of the system file may be difficult to synchronize. This problem is solved if every user or group has control over their own part of the system file. The multi-user support in SDL Suite is based on letting the users split the system file into several files, called *control unit files*. Where to split the system is an active action taken by the user, but the idea is to partition the system according to work responsibilities and to assign a control unit to each partition. Apart from supporting parallel development, the control unit files also facilitate merging of the individual results to one common system file. An example is a large SDL system where there are several blocks on system level developed in parallel by different developers. Each block could then be associated with a control unit file. Now, the blocks can be updated independently and the changes are shaping the local control unit files. Also, there is no need to manually merge changes into the system file when the work has been done. This is because the management of control unit files, performed by the Organizer, includes the merge.

Another way to get support for parallel development is through using the object oriented concepts of SDL, i.e. types and packages. Instead of introducing a block reference symbol for every subsystem we introduce a block type. The block types are placed in packages together with the signal and data type definitions that are needed within that particular block. If two blocks need the same signals and data types then these definitions are placed in a package of their own. Other packages can then use this package. In the same way as we define block types we can also define process types instead of process reference symbols. The different packages containing the types can be analyzed independently. We can also create instances of the types to simulate, verify and validate the behavior of a particular part of the system. When all parts of the system have been designed and tested we put them together, i.e. we create instances of the types and test and verify the overall system behavior.

Prestudy/Conceptualization Phase

The prestudy/conceptualization is the initial activity needed to establish the core requirements that satisfy a well-defined user need and to identify the risks involved in the project by establishing a proof of concept for the project. It is not in the scope of the prestudy/conceptualization to completely analyze all aspects of the requirements, it should rather establish a vision of the underlying idea. It should also validate the assumptions on user needs and implementation technology. One way to do this can be to build an early prototype that illustrates the goals of the project. The prototype may also be an aid in the identification of risks, e.g. related to technical aspects or to the functionality of the system.

In many cases the prestudy/conceptualization can be viewed as a project by itself, following essentially the same steps and phases as any other project, with the exception that the time scale is shorter and that the result is not a finished system. The tangible results from the prestudy/conceptualization should be:

- A clear vision of the requirements (possibly illustrated by an early prototype)
- A risk assessment that identifies the major obstacles ahead

Requirements Analysis Phase

After the prestudy/conceptualization phase it is time for the first iteration through the activities in the SOMT method. This will go through the requirements analysis, system analysis and system/object design and sometimes also the implementation activity. This first iteration is in some sense the most important part of the project since it involves both an analysis of the requirements on the system and the development of an architecture that will meet these (and future) requirements. If this first iteration is successful the further elaboration and refinement of the system will work without any catastrophes. If the first iteration is not successful, e.g. an insufficient understanding of the requirements leading to an inappropriate system architecture, there are loads of problems ahead.

The requirements analysis phase starts the first iteration. The activities of this phase are of course centered around the analysis of the requirements as detailed in chapter 71, *Requirements Analysis*. When this phase is finished the analysis team must have established a common vo-

cabulary and a common understanding of the system's desired behavior. The deliverables are first versions of the models associated with the requirements analysis:

- The textual requirements
- The data dictionary
- The requirements object model, including both information models of the problem and context diagrams
- The use case model
- The system operations model

The most important of these are the use case model, the requirements object model, including the context diagrams, and the data dictionary. Together these describe the major behavior of the system and define a vocabulary describing the application and its environment. The textual requirements are hopefully part of the input to the requirements phase and it is often useful to elaborate these further mainly in the case where the original requirements are too unstructured or incomplete. The system operations are useful if the level of abstraction in the use cases are not found to be detailed enough.

In practise most of the time during the requirements analysis is usually spent designing the use cases, using these as a means to investigate the intended behavior of the system. It is impossible to completely cover the entire set of behaviors in the requirements analysis. Finding the right trade-off between the completeness of the use cases and the time and effort spent in requirements analysis is an important aspect of this phase. A recommended practise from [33] is that the use cases should cover 'approximately 80% of the primary scenarios along with a representative selection of the secondary ones'. The use case model (as well as the other models) will be further extended and refined during the elaboration phase so it is important to force the requirements analysis phase to a completion when a sufficient level of understanding has been reached.

System Analysis Phase

The system analysis phase takes over where the requirements analysis stopped. Where the requirements analysis focussed on *external requirements* the system analysis focuses on the *internal architecture* of the system. The analysis object model is the major medium used to describe the architecture. The architecture is defined in terms of the objects and object structures defined in this model. To establish the responsibilities of the different classes and verify that the proposed architecture is sufficient the use cases from the requirements analysis are in this phase refined in the analysis use case model.

It is not uncommon in practise that there is an overlap in time between the different phases and this is also true for the requirements analysis phase and the system analyses phase. It makes sense to start sketching the architecture of the system before the requirements analysis is brought to completion to capture ideas about the architecture whenever they emerge. It is however important to view this as a side activity and not forget to force the requirements analysis to a completion before digging into the details of the architecture.

The system analysis is finished when there is a consistent and well understood definition of the major objects needed in the system, including a clear definition of the responsibilities for the different objects, and the architecture is validated using the analysis use cases. It is beneficial if the system analysis can be performed by a small team consisting of the best people available including an experienced architect and some more developers.

System Design Phase

For many projects there is a need to divide the design work on more than one design team. In this situation the system design phase is a crucial phase since this is when the interfaces between the different parts of the system are defined. A good interface definition is essential to give the different development teams a possibility to progress with their work without confusion. In the system design the logical architecture from the system analysis is used to decompose the system into subsystems to be implemented by different teams and the interfaces between the subsystems are defined. The static interface is defined in terms of signals and/or remote procedure calls. The dynamic aspects of the interface are

defined in the design use cases that should be precise enough to be used as test cases by the design teams.

For practical reasons it is best if the same team that performed the system analysis phase also is involved in the system design phase, but extended with more designers and test engineers from the teams that will continue the design work to ensure that an understanding of the architecture is spread also into the different design teams.

The system design also includes planning the design module structure to be used in the design of the system to facilitate a smooth integration between the different subsystems. The design module structure is in essence a definition of the structure of the work to be performed by the different teams and it must be designed in order to facilitate reuse of common components and existing frameworks.

A third aspect to be considered in the system design phase, which is more of a project management nature, is to plan the following design and elaboration phases. The planning should be based on the achieved knowledge of the architecture and behavioral requirements and is intended to define the order in which the different features of the application is designed. The major criteria for defining this order is to minimize the risks involved in the project by validating the assumptions about the architecture as soon as possible and by designing the difficult and uncertain parts as early as possible. This implies that the design phase following the system analysis should concentrate on designing a first version of the application that covers an ‘interesting’ subset of the behavioral requirements (as defined in the use cases) and create a skeleton structure where all major components are in place but where their complete functionality is not yet realized.

When the system design is completed the tangible results are thus:

- A detailed architecture described using SDL block structures, including:
  - Static interface definitions with signals, remote procedures and the necessary data definitions, possible described using ASN.1
  - Design use cases describing the dynamic aspects of the interfaces
- A design module structure describing the structure that is to be used in the design phase

- A plan for the design process, that identifies key concepts to be designed and key use cases to use as a focus of the design phase and the elaboration phase

Design/Implementation Phase

The design/implementation phase, which as discussed above in general, is performed in parallel by several different development teams is in an SDL based SOMT project aimed at producing an executable SDL model of the application. Executable may here mean that it is possible to verify the design use cases against the SDL-model by executing the SDL model in e.g. a simulator or state space exploration based tool, but it may also imply an implementation activity that creates a first version of the application running in the target environment.

The choice of when to start the implementation activity is mainly risk-driven. If performance or memory requirements are crucial and seen as a risk in the project, e.g. if the hardware architecture is new and untested, then at least parts of the system should be implemented on the target to minimize these risks. If this is not the case, the integration with the target hardware can be planned to be part of some of the iterations in the elaboration phase.

Depending on the circumstances for the specific project the target platform can of course be very different, from the simplest case being that the target platform is the same type of computer as the development platform to the more complex situation where an embedded system is to be built and the target platform is a small micro processor. In the latter case, the implementation activity for an SDL-based project does in general consist of using a code generator to generate the code from the SDL system, write adaptation code to interface to the target hardware and the run-time environment to be used on this platform, and to use a cross compiler to produce code that is executable on the target machine.

In any case the deliverable from the design/implementation phase is an executable system that implements some of the requirements posed on the application and that is internally released as the first version of the application. The system should also have been through a testing process that verifies that the selected design use cases that where implemented indeed works as they should. This testing will of course consist of both a module testing performed on the different parts by themselves as well

as a system/integration test that verifies that the system as a whole works as it should.

The Elaboration Phase

The rest of the project, the elaboration phase, consists of a number of iterations through all the activities, from requirements analysis to implementation. Each iteration produces a new internal release of the application and refines and extends the architecture and system until it finally is finished. A typical iteration starts with a requirements analysis activity that includes a review and refinement of the use cases to be implemented in the current iteration. This is followed by system analysis and design activities that checks that the use cases are correctly distributed over the architecture and that the interface definitions are updated if needed. These activities should preferably be performed by an architecture group containing representatives both from the original architecture group who worked in the system analysis/design phase and from the different design teams.

The design and implementation activities that are part of an iteration are of course as before performed by the different design teams, each one refining and extending their part of the system to implement the use cases that are the goal of this particular iteration. The result of the design and implementation activity should be a new internal release of the application that has been through the appropriate module and system tests to verify its functionality.

In addition to extending the functionality of the system each iteration should also be focussed on minimizing the risks in the project. Crucial and difficult aspects should be tackled as early as possible in the elaboration phase to avoid unpleasant surprises at the end of the project.

Summary

A SOMT project can be described in terms of a number of phases, that describe the calendar time of the project. The phases are introduced as a means to get a visibility of important milestones and deliverables from the project. The various phases of a SOMT project are:

- Prestudy/conceptualization phase
- First iteration
  - Requirements analysis phase
  - System analysis phase
  - System design phase
  - Design/implementation phase
- Elaboration phase
  - Planned consecutive iterations

A graphical view of a possible SOMT project is illustrated in Figure 720.

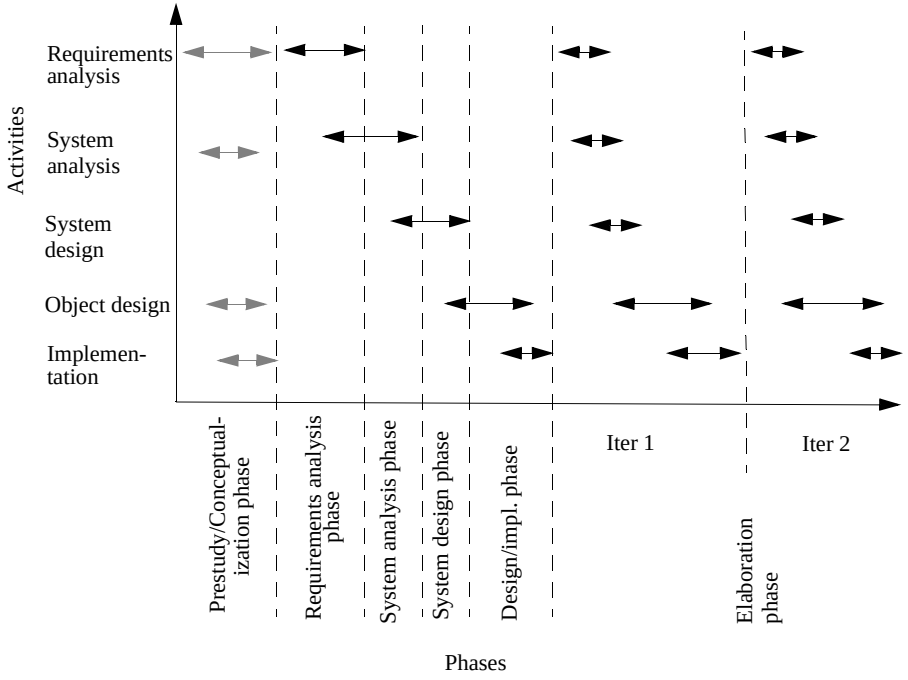


Figure 720: The different phases in a SOMT project

In this figure the various activities (as show on the vertical axis) are divided into a number of phases (as shown in the horizontal axis). The dashed lines indicate the deadlines for the different phases. Note that the activities associated with a specific phase may often start before the deadline of the previous phase is reached. This is common and useful. The purpose of the organization of the activities into phases is not to control when an activity starts, but to be able to focus on when an activity should be finished. This is what indicates the progress of the project.

Chapter

78 *SOMT Tutorial*

This tutorial is intended to present how to combine object-oriented analysis and SDL design in practise in a development process. This is a method developed by IBM Rational, known as the SOMT method, *SDL-oriented Object Modeling Technique*.

We will demonstrate, using an Access Control system as example, the various activities and models in SOMT together with the provided tool support for SOMT in SDL Suite.

Through the tutorial you will practise on various exercises that will get you familiar with the SDL Suite tools as well as the SOMT method.

Introduction

Purpose of This Tutorial

This tutorial presents how to use the SOMT method and the SDL Suite in practise in a design process.

The working example is an Access Control system. The system shall control the entrances to an office. Each employee working in the office has a card and a personal code. To enter the office, the employee enters a card into a card reader and types a personal code on a keypad. To exit the office the employee presses an exit button.

You will perform the development process for the Access Control system applying the SOMT method. The tutorial will guide you through the development process step by step presenting a number of hands on exercises for you to perform. The tutorial is expected to be read sequentially.

After reading the tutorial, you should have gained knowledge about how to apply the SOMT method on a development process.

Note: Platform differences

This tutorial, and the others that are possible to run on both the UNIX and Windows platform, are described in a way common to both platforms. In case there are differences between the platforms, this is indicated by texts like “on UNIX”, “Windows only”, etc. When such platform indicators are found, please pay attention only to the instructions for the platform you are running on.

Normally, screen shots will only be shown for one of the platforms, provided they contain the same information for both platforms. This means that **the layout and appearance of screen shots may differ** slightly from what you see when running SDL Suite on your platform. Only if a screen shot differ in an important aspect between the platforms, two separate screen shots will be shown.

Required Skills

It is assumed that you have a basic knowledge about UML and SDL. We also recommend newcomers to acquaint themselves with the basic features of the SDL Suite tools. You can do this by practising on the exercises in the tutorials provided for the different tools. Please see the previous chapters in this volume.

It is recommended that you have read the SOMT Methodology Guidelines starting in chapter 69 in the User's Manual.

Preparations

1. Make a new empty directory of your own for the purpose of this tutorial, e.g. `~/somttutorial` **(on UNIX)** or `C:\IBM\Rational\SDL_TTCN_Suite6.3\work\somttutorial` **(in Windows)**.

2. Copy the SOMT tutorial directory and its subdirectories in `$telelogic/sdt/examples/somttutorial` **(on UNIX)**, or `C:\IBM\Rational\SDL_TTCN_Suite6.3\sdt\examples\somttutorial` **(in Windows)**, into this new directory (contact if necessary your system manager).

> **Note: Installation directory**
>
> **On UNIX, the installation directory is pointed out by the environment variable** `$telelogic`. If this variable is not set in your UNIX environment, you should ask your system manager or the person responsible for the SDL Suite environment at your site for instructions on how to set this variable correctly.
>
> **In Windows, the installation directory is assumed to be** `C:\IBM\Rational\SDL_TTCN_Suite6.3` **throughout this tutorial.** If you cannot find this directory on your PC, you should ask your system manager or the person responsible for the SDL Suite environment at your site for the correct path to the installation directory.

3. **On UNIX**, `cd` to your own subdirectory `somttutorial`

4. Start SDL Suite.

5. Specify the source directory for the system by double clicking on the *Source directory* symbol located second uppermost in the Organizer window. The source directory specifies where new documents that you have created are saved by default, and from where to read when opening and converting documents. Since there are multiple versions of the Access Control system, each version with diagrams stored on files with identical names (but in different directories), omitting to specify the source directory may cause the wrong version of a file to be opened.

6. In the Set Directories dialog that is opened, select the third radio button associated with *Source directory*. In the text field, enter the complete path and name of your own `somttutorial` directory, if it is not there already. Press *OK* to close the dialog. (You do not have to change any of the other options in this dialog.)

Preparing the Documentation Structure

What You Will Learn

- To prepare a SOMT project by making preparations in the Organizer

Introduction to the Exercise

Your task is to modify the *basic view* of the Organizer to get the desired document structure.

The result of the exercise will be an Organizer structure containing a number of *chapters* and *modules*, see Figure 723 on page 3982. The chapters will correspond to the different activities in SOMT and the modules will correspond to the models in each activity.

Deleting Unwanted Chapters

When you start a new project in SDL Suite you will get the default *basic Organizer view*, see Figure 721. (This view could be different if you do not have the default preferences set). The Organizer contains a few black lines with attached names, the *chapters*. The purpose of these chapters is to group together collections of documents.

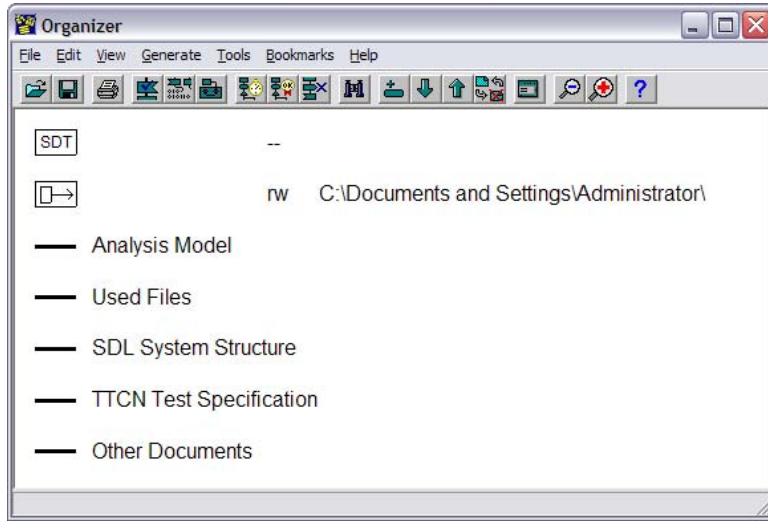


Figure 721: The basic Organizer view

We want each chapter in the Organizer view to represent an activity in SOMT. The current chapters in the Organizer will not fit into our future documentation structure so they should be removed.

Delete the unwanted chapters by following the steps below:

1. Make sure you have the basic view in the Organizer.
2. Select the chapter named `Analysis Model`.
3. Select the *Remove* command in the *Edit* menu or press the `<Delete>` button. You also find the *Remove* command in the pop up menu. The Remove dialog is issued asking you to *Remove* or to *Cancel* the action.
4. Press the *Remove* button. The dialog disappears and the chapter is deleted.
5. Repeat the steps above and remove all of the remaining chapters.

Adding New Chapters

You should now organize the Organizer view into chapters corresponding to the different activities in the SOMT method, i.e. each chapter should contain documents and diagrams from one particular activity.

You will have to add four chapters and they will be named `Requirements Documents`, `System Analysis Documents`, `System Design Documents` and `Object Design Documents`, respectively.

First, add the `Requirements Documents` chapter:

1. Select the *Add New* command in the *Edit* menu. The Add New dialog arises with the *Organizer* radio button set.
2. Select the *Chapter* option in the option menu connected to the Organizer radio button.
3. Change the document name `Untitled` to **`Requirements Documents`**.
4. Press the *OK* button or <Return>. A chapter named `Requirements Documents` will appear as the uppermost chapter object.
5. Now repeat the steps above and add the three remaining chapters and name them **`System Analysis Documents`**, **`System Design Documents`** and **`Object Design Documents`**, respectively.

If the chapters show up in another order than the one you want in the Organizer window, you may move a selected chapter by using the arrow quick buttons in the tool bar.

6. If needed, move the chapters in the Organizer to get a structure corresponding to the one in Figure 722.

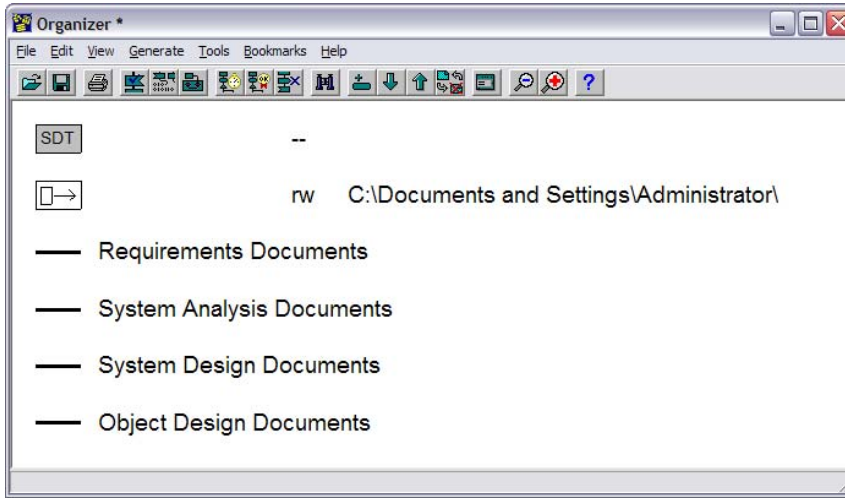


Figure 722: The Chapter structure

Adding the Organizer Modules

The next step to take when preparing the document structure is to add the Organizer *modules*. A module in the Organizer forms a naming scope around the documents it contains. It may contain any kind of documents.

As each activity in SOMT consists of a number of models, it seems natural to let a model correspond to a module in the corresponding chapter. You should now add the modules to the chapters in the Organizer structure.

1. Select the chapter named `Requirements Documents`.
2. Select the *Add New* command in the *Edit* menu.
3. In the Add New dialog, make sure that the Organizer radio button is set. Select the *Module* option in the Organizer option menu.
4. Change the name `Untitled` to **`RequirementsUseCaseModel`**

> **Note:**
>
> You are not allowed to have any space characters in the name of a module.

5. Press the *OK* button. A module named `RequirementsUseCaseModel` appears in the `Requirements Documents` chapter.

6. Now add the other modules to their respective chapter in the Organizer view. Let each model in a SOMT activity have its own module. The document structure in the Organizer should look like Figure 723 when you are finished.

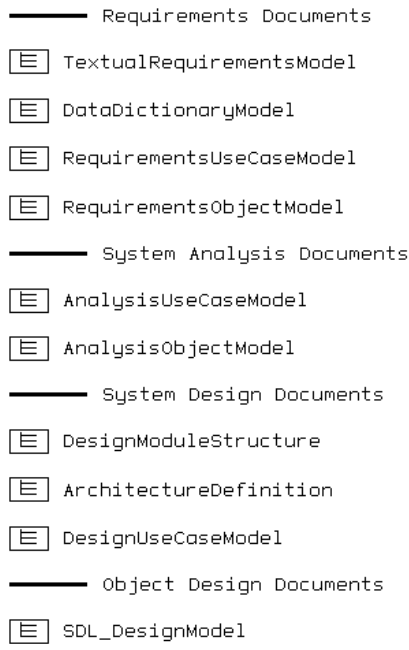


Figure 723: The complete document structure

This structure will form the framework to organize the forthcoming documents around.

7. *Save* the Organizer structure and name the file **`accesscontrol.sdt`**.

Now you have finished the preparations and you can start to develop the Access Control system using the SOMT method.

Identifying the Requirements

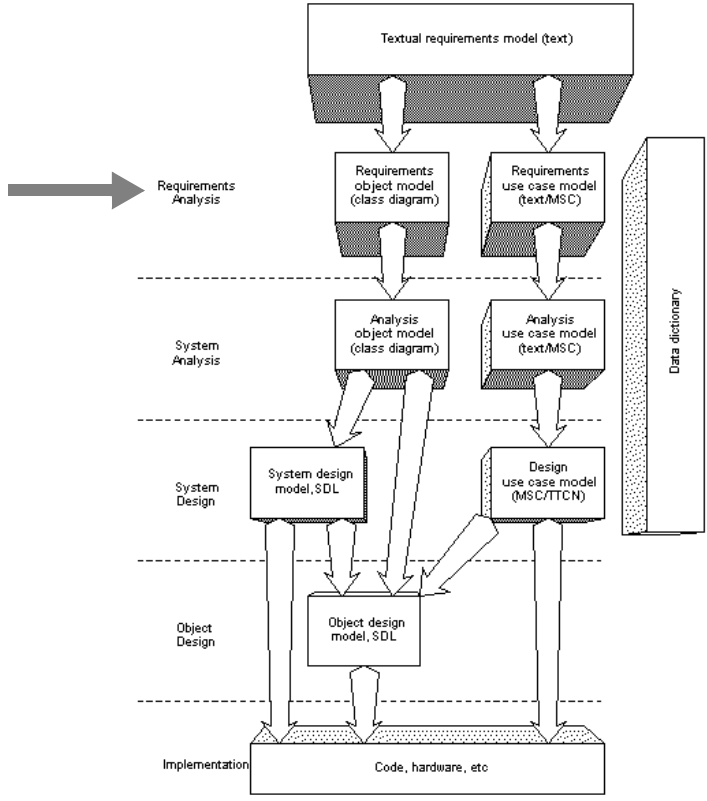


Figure 724: Overview of the SOMT process

What You Will Learn

- To bring in external (requirements) documents into the Organizer
- To identify important concepts
- To use a data dictionary
- To identify actors and use cases and to compile the information gained into textual documents

- To create a textual use case
- To create an MSC use case out of a textual use case
- To make a requirements object model
- To connect important concepts in the different documents with *implinks*
- To perform consistency checks

Introduction to the Exercise

In this exercise you will perform the tasks associated with the requirements analysis activity. The purpose of the requirements analysis is to:

- Gain understanding of the problem domain - the Access Control system and the environment in which it is going to exist.
- Find and understand all requirements imposed on the Access Control system.

Producing a complete requirements analysis would take too much time in this tutorial. Therefore, you will only perform parts of every required step of the process.

The result will not be a complete requirements structure, but you will have acquired knowledge of how to use the SOMT method in the process of identifying requirements.

Preparing the Exercise

You can use your own document structure from the previous exercise (just move your `accesscontrol.sdt` file to the `ReqA` directory), or use a provided solution.

1. Open the system file `somttutorial/ReqA/accesscontrol.sdt` **(on UNIX)**, or `somttutorial\reqa\accesscontrol.sdt` **(in Windows)**.
2. Check that the Source directory is set to `somttutorial/ReqA/` **(on UNIX)**, or `somttutorial\reqa\` **(in Windows)**, in the same way as you did in the preparation to this tutorial (see "Preparations" on page 3977).

Studying the Textual Requirements

Including External Textual Requirements

A textual document with requirements is the input to the Access Control system development project and it will form the base from which the Access Control system is developed. You will later on create implementation links (so called *implinks*) between the textual requirements document and other models. This is done to make it possible to follow a requirement through a number of models all the way down to code.

The textual requirements document of the Access Control system is contained in a text file. This file should now be included in the Organizer work area.

1. Select the module named `TextualRequirementsModel` in the `Requirements Documents` chapter.
2. Select the *Add Existing* command in the *Edit* menu.
3. In the Add Existing dialog, change the filter to **`*.txt`** and press the *Filter* button. Select the file `TextualRequirements.txt` and press *OK* to add it.
4. The `TextualRequirements` document is now added to the module `TextualRequirementsModel` in the Organizer and the *Text Editor* showing the document is opened. The document looks like Example 651.

Example 651: The textual requirements ––––––––––––––––––––––––

The task is to design the software to support a computerized Access Control system. The purpose of the system is to control the accesses to an office.

An entrance leading to an office can have four different security levels:

1. Always unlocked
2. Requires a card to unlock
3. Requires a card as well as a code to unlock
4. Always locked

The security levels of an office entrance can be altered during the day.

Each employee working in the office has a card with a personal code consisting of four digits. To open a door with security level three, the

employee enters her card into a card reader and types her personal code on a keypad. The time between consecutive keystrokes when typing the code is not allowed to exceed three seconds. To enter through a door with security level two, the employee just enters her card into a card reader.

Each entrance leading into the office consists of a door with an electric lock as well as a card reader, a keypad and a display on the outside, and an exit button on the inside. The employee needs a card and a code to *enter* the office. To exit, the employee just presses the exit button and the door is unlocked for ten seconds.

All entrances communicate directly with a central controller which makes sure that a validation of the correctness of cards and codes is performed. The controller has access to a database consisting of all card numbers and their corresponding personal codes. If the card is valid and, in case of security level three, the corresponding code correct, the door is unlocked for ten seconds and the employee may enter. In case of an invalid or unregistered card, access to the office is not allowed. In case of an incorrect code, the employee is informed of this and must try again by entering the card into the card reader and retyping the personal code.

The Access Control system must read its data, consisting of card numbers with their corresponding personal code, from a database. The database is managed by using a separate management system that is not developed within the project. The system operator, who is running the management system, is authorized to register new employees, cards and codes, to change a code if the employee wishes so, to delete employees from the database and to change the security level of an entrance. The system operator is also responsible for initializing the Access Control system. All the actions mentioned above are done using the management system.

The system must be able to recover from computer and connection failures. If a connection between an entrance and the central controller is lost, the door is locked from the outside not permitting anyone to enter (i.e. security level four is set). It is, however, possible to open the door from the inside by means of the exit button.

The system must be extensible to include new functions and be easily maintained.

Creating Textual Endpoints

Now you should study the textual requirements document and mark all concepts (nouns) that you find essential for the problem domain as *link endpoints*. These marks will be very useful in later stages of the project. In this tutorial most of the endpoints in the textual requirements document have already been created. You task is to add the two missing ones:

1. In the second sentence of the textual requirements document locate the word “office” and mark it with the mouse.

   When you mark an endpoint see to it that you only mark the word itself and not any additional characters, like a space or a dot after the word.

2. In the *Link* submenu in the *Tools* menu, choose *Create Endpoint*.

3. The text will be underlined indicating that the text fragment now is a link endpoint.

4. Now, locate the word “entrance” in the third sentence and create an endpoint out of it by repeating the procedure above.

5. If you go through the rest of the document you can see that the rest of the important concepts already have been marked as endpoints.

6. *Save* the document.

7. Open the *Link Manager*. This is done by choosing *Link Manager* in the *Link* submenu in the *Tools* menu. You can do this either in the Organizer window or in the Text Editor window; the result will be the same.

   The Link Manager window will pop up showing all the endpoints of the textual requirements document. The endpoint background color is used to show the endpoint status. As the endpoints are newly created, and the link file has not been saved yet, the background of the endpoints is painted gray.

8. *Save* the link file from the *File* menu, giving it the name `Links.sli`.

9. Close the Link Manager window.

Creating the Data Dictionary

A data dictionary is a textual document which should define all important concepts found during the whole development process. It forms a common vocabulary for the members of the project. It is a good idea to:

- Provide each item included in the data dictionary with a name and a brief explanation.
- Categorize the concepts in nouns, verb phrases and relation phrases.
- Sort the concepts alphabetically.
- Have a section in the data dictionary for each activity. This might be a good idea because a certain concept often has different meanings in different activities. For example, a concept can be described by a class in one activity and in the next activity it might be described by a block with a corresponding process.

All the important objects, relations and verbs that you find in the textual requirements should be included in the data dictionary. This has already been done in an existing `DataDictionary` file, so you do not have to do anything. Just add the existing file:

1. Add the existing `DataDictionary.txt` file to the `DataDictionaryModel` module in the Organizer. The Text Editor will show the `DataDictionary`.
2. Read through the document to get yourself acquainted with the problem domain vocabulary.

All nouns, relation phrases and verb phrases in the data dictionary are marked as link endpoints. This has been done to make it possible to do *entity matches* between any model and the data dictionary. An entity match checks that all entities in one model have matching entities in another model. That is, we can check that all entities in a model really are described in the data dictionary. This will be performed in "Entity Match" on page 4005.

The example below shows a part of the requirements analysis data dictionary.

Example 652: A data dictionary

Nouns/Objects

Access control system - A system to control the access rights to an office so that no unauthorized persons can enter without permission.

Card - Each employee working in the office gets a card and a corresponding personal code. By means of this card and code, the employee can get access to the office.

Cardnumber - The number that uniquely defines a card.

...

Relation Phrases

Card with code - Each employee in the office has a card with a personal code.

Connection between central controller and entrance - There is a connection between every entrance and the central controller.

...

Verb Phrases

Change code - An operation done by the system operator to change the code of a card.

Change Security Level - An operation done by the system operator to alter the security level of an entrance.

Connection is lost - The connection between an entrance and the central controller can sometimes fail. In case of broken connection nobody can enter the office. It is, however, possible to leave the office.

...

Creating the Use Case Model

The purpose of a *use case model* is to capture the requirements and present them from the users point of view, thus, making it easier for the intended users to validate the correctness of the requirements analysis.

The use case model consists of:

- A list of actors
- A list of use cases
- A number of MSCs (message sequence charts) and/or textual use cases

The use case model is also a useful source of information when developing the *requirements object model*, see "Creating the Requirements Object Model" on page 4001.

A use case is a sequence of actions showing a possible usage of a system. Use cases developed during the requirements analysis activity should mainly concern the interaction between the system and the users of the system. No message exchanges within the system should be shown.

Users of a system may be people, other systems or objects outside the system border which interact with the system.

An *actor* is a user taking part in a use case. An actor is not supposed to be an individual user, but rather represents one of the different roles a user can play when interacting with the system.

There are different ways to describe a use case:

- A textual description of the use case
- A description of the use case using an MSC
- A combination of both a textual description and an MSC

Describing use cases using textual descriptions will make it easier to model exceptions and alternative paths of action sequences. Describing use cases using MSCs will make the use cases more formal and easier to verify. Also, as MSCs will be used in the coming activities, it might be a good idea to start using them already in the requirements analysis. The tutorial will use both textual descriptions and MSCs in the requirements analysis activity, and only MSCs in the later activities.

Creating a List of Actors

Now it is time to create a list of actors. The list of actors should list the actors by name, together with their respective responsibility.

1. In the Organizer, select the `RequirementsUseCaseModel` module and choose *Add New*. In the Add New dialog, set the *Text* radio button and choose *Plain* in the corresponding option menu. Name the new document **`ActorsList`** and set the toggle button *Show in Editor*. This will give you a new text document in the Organizer window and an empty Text Editor window will pop up.
2. Try to find the actors of the Access Control system by studying the textual requirements in Example 651 on page 3985. For information on how to find actors, see "Finding Actors" on page 3991 below.
3. List the actors by name in the newly created textual document together with a brief description of the actor's role when interacting with the system.
4. Mark the actors in the `ActorsList` as endpoints in the same way as you did in "Creating Textual Endpoints" on page 3987.
5. *Save* the document giving it the name **`ActorsList.txt`**.

Finding Actors

You will find actors by studying the textual requirements. Useful questions to ask are:

- Which users will need services from the system to perform their tasks?
- Which users are needed by the system to perform its tasks?
- Are there any external systems that use or are being used by our system?

In practise, the activity of defining actors should be performed iteratively. Try to find as many of the actors as possible now. If you do not believe you found them all, start creating some MSC use cases (as having done some MSCs often makes it easier to determine the actors). Then go back and complete the list of actors.

In our case with the Access Control system we find that an employee, who daily interacts with the system, is an obvious candidate for the list

of actors. Also, considering the third question above, it is obvious that the management system is being used by our system and therefore should be added to the list. The third actor, the door, may not be so easy to find at a first glance, but when you have created the MSCs it will become more evident that the door is an actor as well. The door's interaction with the system consists of notifying the system every time it is opened or closed.

The example below shows a part of the list of actors.

Example 653: Part of a list of Actors

Employee - Someone who needs to enter and exit the office. To enter the office, an employee must have a registered card and (depending on the current security level) a corresponding personal code. To exit, the employee must press an exit button to unlock the door.

ManagementSystem - The management system starts and maintains the Access Control system. All changes to the database are handled by the management system. The management system is run by a system operator.

...

Creating a List of Use Cases

When you have defined the set of actors it is time to describe the way they interact with the system, which is done in use cases. The first step is to create a list of all use cases. The list of use cases should list the use cases by name together with a short description.

1. *Add* a *new* plain text document in the `RequirementsUseCaseModel` module. Name it **`UseCaseList`** and set the toggle button *Show in Editor*. Press *OK*.

2. Try to find the normal use cases and list them in the newly created textual document. For information on how to find use cases, see "Finding Use Cases" on page 3993.

3. To each use case, add a general one-sentence description of its functionality.

4. For each normal use case, examine which exceptions that can occur and state these exceptions as well in the list.
5. Mark the use case names in the `UseCaseList` as endpoints.
6. Save the document giving it the name **`UseCaseList.txt`**.

Finding Use Cases

It is often quite easy to identify use cases by looking at the purpose of the system. To verify that you have identified most of the important use cases you should:

- look at the list of actors, and, for each actor,
- identify the tasks that the actor should be able to perform and the tasks which the system needs the actor to perform. Each such task is a candidate for a new use case. It is often very useful to check the textual requirements document for verb phrases (or you could look directly in the DataDictionary to see which verb phrases that have been stated as important); these are possible candidates for use cases.

Start with the employee actor and try to determine which actions he or she needs to perform. There are different ways to enter an office, either using a card or using both a card and a code. Both ways are obvious candidates for the use case list. Also, the employee must be able to exit the office, this will be yet another use case.

The Management system must inform the Access Control system when there has been a change in security level. This will be our fourth use case.

As for the door actor, the task of notifying the system when a door is opened and closed can be included in the enter/exit office use cases. You should always try to make the use cases as complete as possible, that is, make **one** complete use case instead of several minor ones.

When you have found the normal use cases, refine them by examining the exceptions that are possible for each use case. Look in the textual requirements document and try to find the exceptions that can occur.

In the case where an employee enters the office, the first thing that can go wrong is that there is a connection failure between the entrance and the central controller. Other possible things that can fail are that the card

is invalid, the code is wrong, the time between consequent keystrokes when typing the code is too long, and, finally, the door is never opened even though it was unlocked. All these exceptional cases can be found by studying the textual requirements thoroughly.

The example below shows a part of the use case list.

Example 654: Part of a Use Case List ––––––––––––––––––––––––––

Normal Cases:

Enter_Office_With_Card - Describes the interaction between an employee and the Access Control system when the employee wants to enter the office through a door with security level two.

Enter_Office_With_Card_And_Code - Describes the interaction between an employee and the Access Control system when the employee wants to enter the office through a door with security level three.

Exit_Office - Describes the interaction between an employee and the Access Control system when the employee wants to exit the office.

...

Exceptional Cases:

Exc_No_Connection

Exc_Invalid_Card

...

––

Creating a Textual Use Case

Now that we have a list of the actors to the system as well as a list of use cases, we can start to create a more detailed description of the use cases. A textual use case consists essentially of natural text structured into a number of text fields, see "Describing a Textual Use Case" on page 3995. In this exercise we will only create one textual use case, as creating them all takes too much time. The use case we will focus on

throughout the rest of the tutorial is the one where an employee enters an office with both a card and a code.

1. Add a new textual document in the `RequirementsUseCaseModel` module and name it **`Enter_Office_With_Card_And_Code`**.
2. Try to create the textual use case consisting of the fields described in "Describing a Textual Use Case" on page 3995.
3. Create endpoints of the textual use case name (for consistency use exactly the same name as you used in the list of use cases) and of the actors involved in the use case.
4. Save the document giving it the name **`Enter_Office_With_Card_And_Code.txt`**.

Describing a Textual Use Case

A textual use case should consists of the following fields:

- **Name**: The name of the use case.
- **Actors**: A list of the actors involved in the use case.
- **Preconditions**: A list of properties that must be true for this use case to take place.
- **Postconditions**: A list of properties that are true when the use case is finished.
- **Description**: A textual description of the normal sequence of events that describe the interaction between the actors and the system.
- **Exceptions**: A list of exceptional interactions that complement the normal flow of events described in the `Description` field. If an exception leads to different postcondition properties compared to the normal sequence this should be noted.

The description field should thus describe what happens when everything is going as expected. No exceptions should be considered here. They are not described until the exceptions field.

The example below shows a textual description of the use case "Enter office with card and code."

Example 655: A Textual Use Case

Use case name: Enter_Office_With_Card_And_Code

Actor: Employee, door

Preconditions: System is initialized, security level three is set, and the door is closed and locked. The display displays “Enter card”.

Postconditions: The door is closed and locked again.

Description: An employee enters a card into the card reader. The display displays “Enter code”. The employee enters a code consisting of four digits using the keypad. The door is unlocked and “Please enter” is displayed. The employee opens the door, enters the office and closes the door again. The door is locked and “Enter card” is displayed.

Exceptions:

- If the employee enters an invalid or unregistered card, “Invalid card” is displayed for three seconds and then “Enter card” is displayed.

- If the time between consequent keystrokes when typing the code exceeds three seconds, everything is interrupted and “Enter card” is displayed.

- If the employee types the wrong code, “Wrong code” is displayed for three seconds and then “Enter card” is displayed.

- If the employee does not open the door within ten seconds after it has been unlocked the door is locked again and “Enter card” is displayed.

- If there is no connection between the entrance and the central controller and a card is entered, then the text “Connection failure” is displayed for three seconds and then “Enter card” is displayed again.

Creating an MSC Use Case

The second notation for use cases used in SOMT is MSCs. Creating MSCs for all the use cases and their exceptions takes too much time in this tutorial. Therefore you will concentrate on the use case corresponding to the textual description you just created,

`Enter_Office_With_Card_And_Code`, and one of its exceptions, when an employee enters an invalid card.

1. Select the module `RequirementsUseCaseModel` and choose *Add New*. In the Add New dialog, set the *MSC* radio button. Name the document **`Enter_Office_With_Card_And_Code`** and set the *Show in Editor* toggle button.

2. In the MSC Editor, try to create the MSC. Look at the textual description of the use case and describe it by means of the notations defined for MSCs. Also, make references to exceptions at the points where these can occur. Figure 725 shows an example of the complete MSC.

   – Each actor should be represented by a separate instance. The Access Control system itself should also be represented by a separate instance.

   – Actions, displayed messages, etc. should be drawn as MSC messages between the instances.

   – An exception is drawn by adding an *MSC reference* symbol, located last in the MSC Editor's symbol menu. An MSC reference symbol is a reference to another MSC, described in a separate MSC diagram. The symbol is added to one of the instance axes. By convention, MSC exceptions are named "exc" followed by the name of the exception. To connect the symbol to all three axes, select *Connect* from the *Edit* menu. Press the *Global* button to connect the reference symbol to all axes.

3. Save the MSC diagram giving it the name **`Enter_Office_With_Card_And_Code.msc`**.

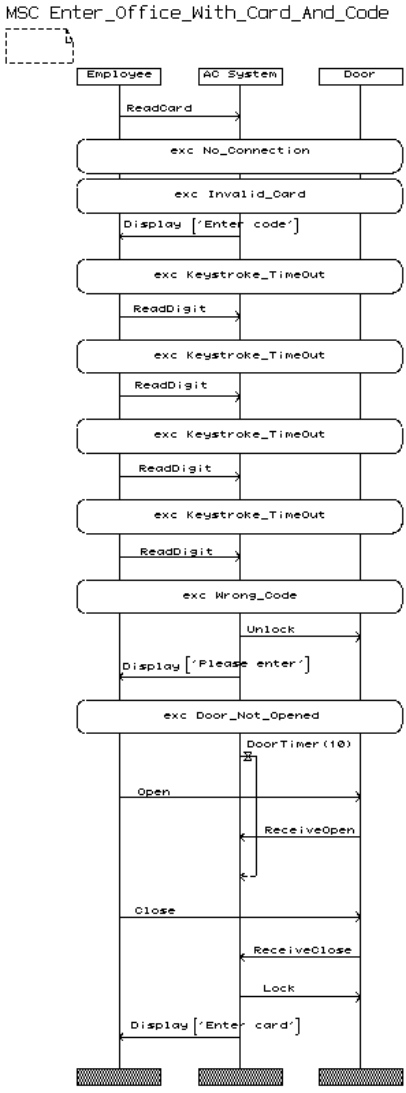


Figure 725: An MSC example

4. Create a new module in the `Requirements Documents` chapter in the Organizer. Name the module **`MSC_Exceptions_ReqA`**.
5. Add a new MSC document to the newly created module in the Organizer and name the document **`Exc_Invalid_Card`**.
6. In the MSC Editor, try to create the exception, i.e. describe what happens when an employee has entered an invalid card.
7. Save the diagram giving it the name **`Exc_Invalid_Card.msc`**.

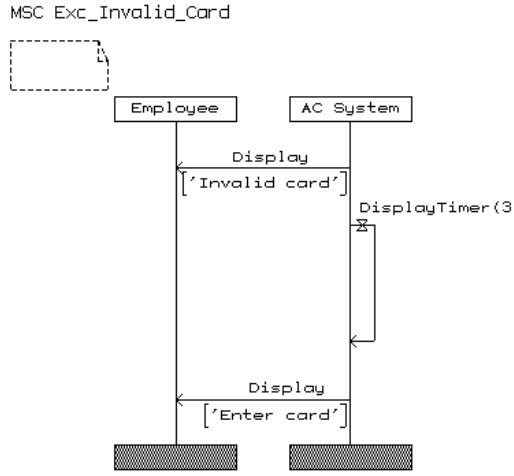


Figure 726: An MSC exception example

8. In the Organizer view, select the MSC `Exc_Invalid_Card` and then choose *Associate* in the *Edit* menu. The Associate dialog appears.
9. Choose to associate the `Exc_Invalid_Card` MSC with the `Enter_Office_With_Card_And_Code` MSC as it is an exception to this use case.

The `Requirements Documents` chapter should now look like in Figure 727.

In reality you repeat the steps above for all the use cases found and associate each one of them with its exceptions. In this tutorial, however,

we will not create the entire use case model as that would take too much time.

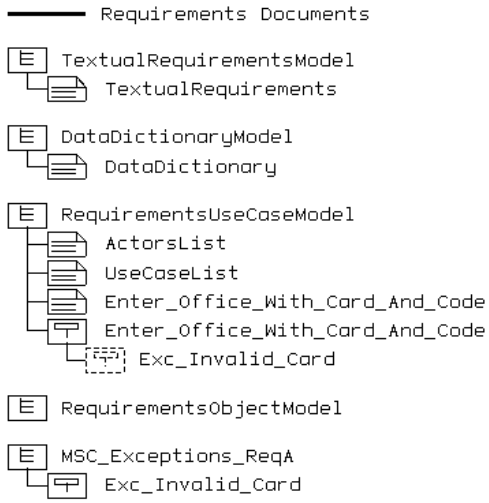


Figure 727: The Requirements Documents chapter

Now, when we have our use cases and a data dictionary we will continue the activity with producing a requirements object model. In practise, you should work with all the models in parallel. The activities in SOMT are **not** supposed to be performed in a sequential order, rather, producing the models is a highly iterative process.

Creating the Requirements Object Model

The requirements object model is intended to capture the objects, the relations between these objects and other concepts of the real world that are of importance for the application we intend to build. There are different types of concepts that can be described in this model. The two major diagram types show the logical structure of the data and information and the context of the system.

Relations between objects in the model will be expressed through associations, aggregations and inheritance.

Creating a Requirements Object Model

Now you should create the requirements object model.

1. In the Organizer, select the `RequirementsObjectModel` module and choose *Add New*. In the Add New dialog, set the *UML* radio button and make sure the *Object Model* option in the UML option menu is set. Name the new document **`LogicalStructure`** and set the toggle button *Show in Editor*. This will pop up an empty OM Editor window.
2. Try to find the objects, see "Identifying the Objects" on page 4003.
3. Enter the classes found into the object model diagram in the OM Editor and give them a suitable name. As you can see, every class is automatically marked as an endpoint.
4. Relate the classes by means of associations, aggregations and inheritance, see "Identifying the Relations" on page 4004.
5. Consider if multiplicity is needed on any of the associations and if so, add it. (Double-click a line to bring up the Line Details dialog.)
6. To increase the readability of the model, name the associations or attach role names to the classes. The diagram should look something like in Figure 728 when you are finished.

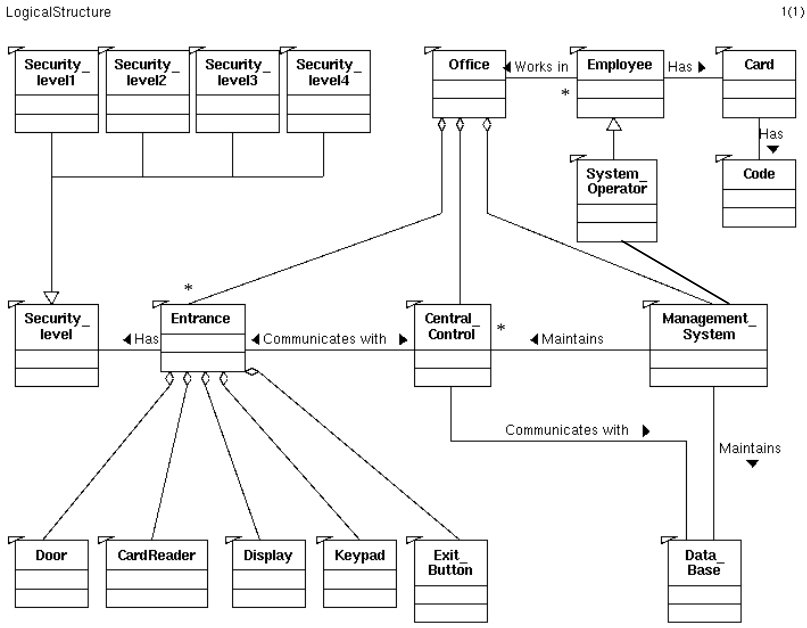


Figure 728: The logical structure

7. Save the diagram giving it the name **`logicalstructure.som`**.

8. Add yet another object model to the `RequirementsObjectModel` module in the Organizer. Name it **`ContextDiagram`**.

9. In the diagram, show the system and the external actors interacting with it. Use collapsed class symbols (select *Collapse* from the *Edit* menu). The classes are automatically marked as endpoints.

10. *Clear* the endpoint on the `Access_Control_System` class as we will not need this. (Select *Clear Endpoint* from the *Link* submenu in the *Tools* menu.)

11. *Save* the diagram giving it the name **`contextdiagram.som`**.

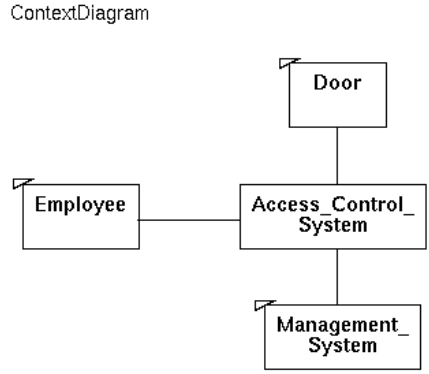


Figure 729: The Context diagram

Identifying the Objects

The main input sources to the requirements object model are the textual requirements, the use case model and the data dictionary. Other sources of information are domain experts, textbooks etc.

A classical way to find the objects is to study the textual requirements and note all nouns (or look directly in the nouns section in the data dictionary). If a particular noun appears in many places, the concept is probably important for the problem domain and should be modeled in the requirements object model.

The use cases are also helpful for finding the objects. They define the actors that interact with the system and these are obvious object candidates. Other likely object candidates are the entities that are transported in to or out of the system. The use cases are helpful in identifying these concepts as well.

The requirements object model should at least describe all concepts that are visible on the outside of the system. This includes all physical entities that a user can see as well as the knowledge a user must have to use the system. It is, however, not only concepts **outside** the system that should be modeled in the requirements object model. Concepts **inside** the system that are so obvious that we know of them already at this stage should be dealt with as well. In our Access Control system, for instance, it is quite easy to see that the system itself is built up of a central control and a number of entrances, each having its own local control. Therefore,

in the requirements object model, we do not model the heart of the system as one class, but as two communicating classes.

Identifying the Relations

The information sources when identifying relations between objects are the same as when identifying the objects. Look for relation phrases in the textual requirements (or use the data dictionary as an information source). You may also take a look at each object and ask the questions:

1. What services does the object provide?
2. Does the object need services from other objects to complete its services?

If the object needs services from other objects, identify these objects and model the relations in the object model.

There are three different types of relations, described below.

The Association Relation

The association relation describes how different classes relate to each other by means of information exchange.

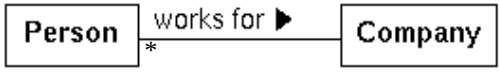


Figure 730: An Association relation

The Aggregation Relation

The aggregation relation is a special case of the association relation and it describes a “consists of” relation. For example: a document **consists of** paragraphs.

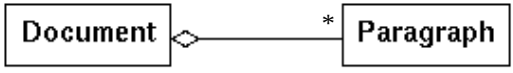


Figure 731: An Aggregation relation

The Inheritance Relation

The inheritance relation describes an “is a” relation. For example: a car **is a** vehicle.

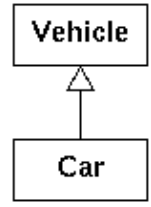


Figure 732: An Inheritance relation

Entity Match

Now it is time to do some consistency checks between the created models. When you do an entity match you check that all entities in one model have matching entities in another model.

1. *Open* the Link Manager. The window popping up shows all endpoints from the models you have created during the requirements analysis activity.

2. To be able to perform an entity match you must be in entity mode (i.e. **not** in endpoint mode). Press the *Show endpoints or entities* button in the Tool bar to change to entity mode. (The view in the Link Manager window will look just the same, since one entity corresponds to exactly one endpoint in all our models.)

The first thing we will check is that all important concepts in the textual requirements model are described in the data dictionary.

3. Choose *Consistency Check* in the *Tools* menu. The Consistency Check dialog appears and you are asked to choose between a link check and an entity match. Set the *entity match* radio button and press *Continue*.

4. A new dialog appears and you are asked to select the documents representing the **from** group. Select the `TextualRequirementsModel` module and press *Continue*. (As you can see, the text document in the module is also selected when you select the module.)

5. Yet another dialog appears asking you to select the documents representing the **to** group. Select the `DataDictionaryModel` module and press *Check*.

6. The Link Manager window will show the result of the entity match in a new *consistency view*. Entities from the **from** group are shown as normal endpoints and entities from the **to** group are shown as dashed endpoints. The links shown are temporary links created by the Link Manager to indicate matching entities, see Figure 733.

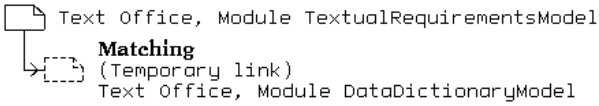


Figure 733: Matching entities

7. As you can see, if you scroll through the Link Manager window, all the concepts from the textual requirements have a matching entity in the data dictionary. (An endpoint from the **to** group without a matching link from it would have indicated that no corresponding entity could be found in the **to** group.)

There are a few more consistency checks which you can perform at this point:

- Check that all entities in the requirements object model are described in the data dictionary.
  - Let the `RequirementsObjectModel` module form the **from** group and the `DataDictionary` module the **to** group.
  - As you can see from the result all concepts but the four different security levels have been described in the data dictionary.
- Check that all important concepts in the textual use cases are described in the data dictionary and in the use case list. Important concepts in a textual use case are the actors and the use case name. The actors should be described in the data dictionary and in the actors list. The use case name should be described in the use case list.
  - In this case the **from** group will be the textual use case, `Enter_Office_With_Card_And_Code`, and the **to** group will be the `DataDictionary`, the `ActorsList` and the

`UseCaseList`. (You can select any number of individual documents in the list, not only modules.)

– The result shows that the use case name was found in the `UseCaseList` and that the actors were described both in the data dictionary and in the list of actors.

- Check that all actors in the use cases are modeled in the context diagram and vice versa.

  – First, the `Actorslist` will form the **from** group and the `Context Diagram` will form the **to** group, then we will do another entity match with the groups vice versa.

Creating Implinks

Now that we know that all our models are consistent, it is time to add the *implinks*. Implinks are used to enable traceability between the models.

We will start with creating implinks from the concepts in the textual requirements to the requirements object model, in particular the logical structure diagram.

1. Open the Link Manager by selecting it in the submenu *Link* in the *Tools* menu if it is not already open.
2. Make sure the window shows endpoint view, not entity view or consistency view. If necessary, press the *Show endpoint or entities* quick button.

   In the Link Manager window you see all the endpoints from the different models in the requirements analysis activity. If you scroll to the very end of the window you can see how many endpoints and links you have in your system. There should be no links at this point.
3. To check that all endpoints are present in the Link Manager window, choose *Check Endpoints* in the *Tools* menu.
4. The Check Endpoints window will pop up showing if any previously unknown endpoints were found. If so, select these and press *Add*. Then press *Continue*.
5. Another version of the Check Endpoints window pops up showing if any invalid endpoints were found. In such case you can choose to

delete these by pressing *Delete*. Press *OK* when you are done and want to close the dialog.

6. To be able to create the implinks between the textual requirements and the classes in the logical structure you only need to see the endpoints from these diagrams in the Link Manager window. You do not have to see all the other endpoints. Therefore, choose *Filter* in the *View* menu.

   The Filter dialog pops up and you can choose to set filter settings for links, endpoint or documents, by selecting from the option menus.

7. Choose to set the filter for *documents* and select that the only documents to be *shown* should be the textual requirements and the logical structure documents.

8. Press *Apply* and then *Done*.

9. In the Link Manager window, highlight the endpoint `Text Office` by first selecting the endpoint and then clicking the *Highlight quick* button in the tool bar. The endpoint is highlighted with a frame around it.

10. Create an implink to the `Class Office` by first selecting the class and then pressing the *Create Link* quick button in the tool bar. The Create Link dialog will open.

11. Name the link **`Implementation Link`** and press the *Create* button. A link from the text “office” to the class Office is created.

The rest of the links between the textual requirements and the object model diagram are created in a similar way. You can do this if you want, or go to the next exercise, where this has been done, and check out the result.

Links from the `UseCaseList` to the different MSCs and their exceptions should also be created. You cannot do this here however, as you have not created all the use cases.

What we aim at here is to create links from the textual requirements, through the object models of the different activities, to the SDL design. Simultaneously we want to create links from the list of use cases, through the use case models in the different activities, to the SDL design. The result of this will be that we can trace a design decision backwards to requirements through either object models or use case models.

Summary

After having completed an entire requirements analysis, the `Requirements Document` chapter the Organizer view should look like in Figure 734.

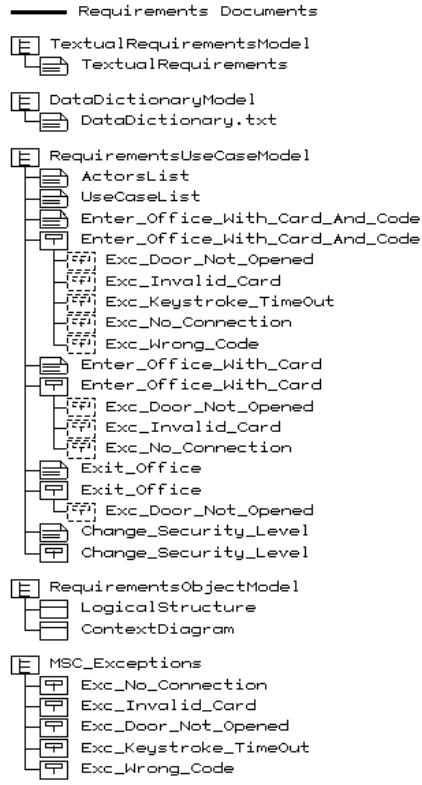


Figure 734: The entire requirements analysis document structure

Performing the System Analysis

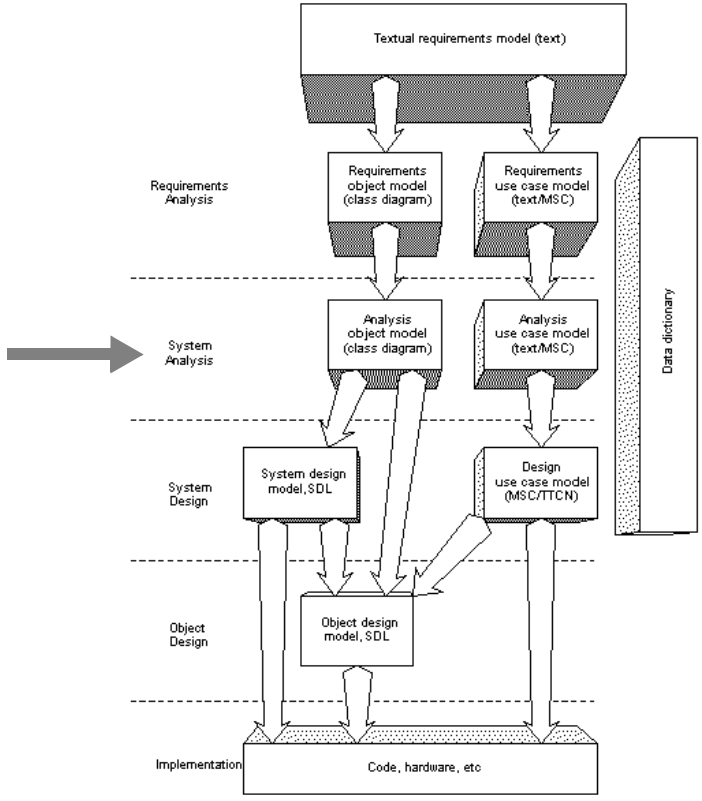


Figure 735: Overview of the SOMT process

What You Will Learn

- To identify and present the logical architecture of a system which includes refining the object model from the previous phase
- To refine use cases from the previous phase
- To use the *Paste As* mechanism

Introduction to the Exercise

In this exercise you will perform the system analysis activity. The purpose of the exercise is to outline a logical model of the Access Control system. This model will fulfil the requirements that were identified in the requirements analysis. In other words, the purpose of this activity is to identify the objects that are needed in the Access Control system and the services these objects should provide.

Producing a complete system analysis structure takes too much time. Thus, you will only perform parts of every step necessary to produce the complete structure.

The input to the system analysis activity is a complete requirements structure with the two main models:

- requirements object model
- requirements use case model

The output from the system analysis activity are the two models:

- analysis object model
- analysis use case model

These two models should be created in parallel through a number of iterations.

Preparing the Exercise

1. Open the system file `somttutorial/SysA/accesscontrol.sdt` **(on UNIX)**, or `somttutorial\sysa\accesscontrol.sdt` **(in Windows)**.
2. Check that the Source directory is set to `somttutorial/SysA/` **(on UNIX)**, or `somttutorial\sysa\` **(in Windows)**.

What you see in the Organizer window is a complete requirements analysis structure with all implinks made.

Creating the Analysis Object Model

The analysis object model is a refinement of the requirements object model. However, when transferring from the requirements analysis to the system analysis you change the focus.

During the requirements analysis the focus is on understanding the problem and the problem domain. In the system analysis the focus is to model a solution and to understand the logical structure of the system that will be the solution to the stated problem. This change of focus should be reflected by the analysis object model.

Little emphasis should be put on implementation aspects during the system analysis activity. Questions regarding the implementation of the solution will very likely hide our actual problem.

A glance at the headlines may give you the impression that the activity of creating the analysis object model is a sequential activity, but it is not. You will probably not first add all the necessary classes, then the relations, and finally specify the attributes and operations. This is, however, the way the text is structured here to make it readable and to highlight the important tasks of the activity.

Creating a Logical Architecture

Now it is time to create the logical architecture diagram.

1. Select the `AnalysisObjectModel` module in the `System Analysis Documents` chapter in the Organizer.
2. Add a new object model diagram and name it **`LogicalArchitecture`**.
3. Open the `LogicalStructure` diagram from the previous activity as well as the new `LogicalArchitecture` diagram in the OM Editor.

Adding Classes to the Logical Architecture

Now you should start adding classes to the logical architecture diagram. For information on how to find the classes, see "Finding Classes" on page 4014.

Several of the classes in the requirements object model can be transferred as-is to the analysis object model. The provided *Paste As* mechanism lets you transfer objects from one model to another while auto-

matically creating implinks between the objects in the two separate models. This mechanism should be used here, see below.

4. *Select* the class `CentralControl` in the `LogicalStructure` diagram and choose *Copy* in the *Edit* menu.
5. Go to the `LogicalArchitecture` diagram by selecting it in the *Diagrams* menu.
6. Choose *Paste As* in the *Edit* menu. The Paste As dialog is opened.
7. Set the option menu to *Class* and see to it that the *Create link* toggle button is **set**. Press the *Paste As* button.
8. Select where in the `LogicalArchitecture` diagram you want to place the `CentralControl` object.

Now you have created a class named `CentralControl` in your `LogicalArchitecture` diagram. The class is connected with an implink to the `CentralControl` class in the `LogicalStructure` diagram. The link is indicated by the filled triangle on the class symbol.

9. Repeat the procedure above with the class `Entrance` but name the new class **`EntranceUnit`** as this is a more descriptive name.
10. The classes `Cardreader`, `Keypad`, `Display` and `ExitButton` are obvious interfaces to our system and, also, parts of an `EntranceUnit`. Repeat the procedure above with these classes. As it is often useful to name objects according to their function, give the new classes names of the form **`xxxInterface`**.

One class in the requirements object model may result in several classes in the analysis object model. One reason may be that the class provides so much functionality that splitting the class into several smaller may be convenient. Another reason may be that a class identified in the requirements object model needs services from other, not yet introduced classes. Consequently these classes should be introduced at this point of the development process.

In practice, the actions in the two cases above are the same. The newly introduced classes should be linked to the original class in the requirements object model. In our example this is the case with the class Door. In the requirements object model we have one single class representing the door. Further analysis, however, shows that the door object includes both a door lock as well as a door sensor. This aggregation structure

should be shown in the analysis object model. See section "Finding Relations" on page 4016.

11. *Copy* the class `Door` in the `LogicalStructure` diagram.
12. *Paste* it three times *as* a class in the `LogicalArchitecture` diagram and see to it that the implinks are created at the same time. *Name* the new classes `DoorUnit`, `DoorLockInterface` and `DoorSensorInterface` respectively.

Now look at the classes you have so far in the `LogicalArchitecture` diagram. Think about the tasks of the different objects. As you can see there is no class that can handle the logic, that is, an object that is responsible for what happens at an entrance. Therefore, you should add such an object and name it **`EntranceCtrl`**, see below. The `EntranceCtrl` will be a part of the `EntranceUnit`.

13. *Copy* the class `Entrance` from the `LogicalStructure` diagram.
14. *Paste* it *as* a class in the `LogicalArchitecture` and rename the class, giving it the name `EntranceCtrl`.
15. Also, *copy* and *paste* the `SecurityLevel` class and its subclasses.

Classes in the requirements object model that only exist outside the system border or classes that do not provide any necessary services should not be transferred at all to the analysis object model.

Finding Classes

Useful sources where you can find objects that may be included in the analysis object model are:

- the requirements object model
- interfaces that the system will have to the environment
- use cases

When you intend to transfer objects from the requirements object model to the analysis object model, consider the following to validate each requirements object:

- Decide if the system needs information about the object to fulfill its task.
- If the answer is "yes" then add the class to the analysis object model.

Another useful way to find the objects is to examine which interfaces the system needs. Often the application area itself makes it obvious what interfaces must exist. To find the interface objects, go through the list of actors and, for each actor, decide which interfaces that are needed to the system.

The use cases from the system analysis activity are also a useful source for finding the objects. The problem is that we have not created these use cases yet. As stated before, the work done in each activity is often done iteratively. This is a typical situation where an iteration is needed as some objects in the analysis object model may not be found until we have created and inspected the analysis use case model.

When you have created the MSCs you should examine them to check which interface objects that are involved and which internal objects that are modified. Also, check if there is a control object that might handle the logic of the use case or if there is a need to introduce such an object in the analysis object model.

Adding Relations to the Logical Architecture

When you have identified all classes it is time to add the relations. For information about how to add relations, see "Finding Relations" on page 4016.

In our Access Control system example, most of the relations from the requirements object model can be preserved.

1. Connect the `CentralControl` to the `EntranceUnit` with an association. Add multiplicity to the association. The `CentralControl` may be connected to several `EntranceUnits` but the Access Control system has only one `CentralControl`.
2. Connect the `EntranceUnit` to the `DoorUnit` with an aggregation. One `EntranceUnit` consists of only one `DoorUnit`.
3. The `DoorUnit` consists of a `DoorLockInterface` and a `DoorSensorInterface`. Add aggregations from the `DoorUnit` to the `DoorLockInterface` and to the `DoorSensorInterface`.
4. Connect the rest of the classes you have added with necessary associations, aggregations and generalizations. Also consider if multiplicity is needed or not.

Finding Relations

Useful sources that may assist the process of finding relations are:

- the requirements object model (preserving and modifying existing relations)
- analysis use case model
- textual requirements

The process of finding new relations and verifying old ones are closely related to the process of creating the analysis use case model. It is mainly a question about which other objects the object needs to know about to be able to provide its services. Also generalizations and aggregational dependencies have to be considered, see "Identifying the Relations" on page 4004.

Adding Attributes to the Logical Architecture

Now we have come to the point where it is time to add the attributes. For information on how to find the attributes, see "Identifying Attributes" on page 4016.

There are not many classes in our `LogicalArchitecture` diagram that need any attributes. In fact there is only one, the class `Display`. This class must be able to display different text messages depending on the situation at hand. Therefore:

1. Define `Text` to be an attribute of the class `DisplayInterface`.

Identifying Attributes

Attributes can be found in:

- the requirements model (keeping existing attributes)
- the textual use cases
- the textual requirements

Attributes describe a property of an object and often correspond to nouns. For example, possible attributes of an object "Person" may be eye color, weight, shoe size, and so on. Attributes that may describe a vehicle are owner, color, current speed, current gear, and direction.

Adding Operations to the Logical Architecture

The last thing to add to the logical structure object model are the operations. For information on how to find the operations, see "Identifying Operations" on page 4018.

Add the operations `Open`, `Close`, `Lock` and `Unlock` to the assembly class `DoorUnit`. This implies that you also must add the operations `Open` and `Close` to the class `DoorSensorInterface` and `Unlock` and `Lock` to the class `DoorLockInterface`.

1. Select the class `DoorUnit`.
2. Click on the operations section in the class.
3. Add the operations `Open`, `Close`, `Lock` and `Unlock`.
4. Repeat the procedure for the classes `DoorLockInterface` and `DoorSensorInterface` adding their respective operations.
5. Continue to add the missing operations of the other classes in the diagram. When you are finished, your diagram should look something like in Figure 736.
6. Save the diagram.

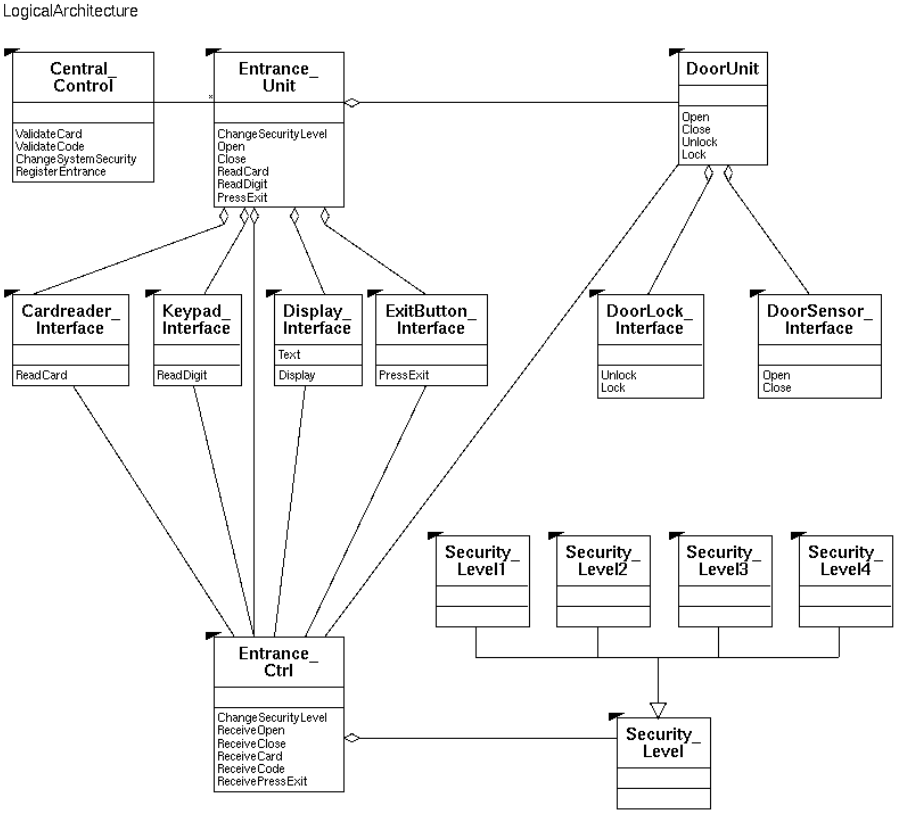


Figure 736: The logical architecture

Identifying Operations

Sources that may support the identification of operations are:

- the requirements object model
- the analysis use case model (the messages in the MSC diagrams)
- the data dictionary (the description of the objects)

By studying the responsibilities of each object is it possible to identify a set of operations that will provide the services assigned to the object.

Let one operation perform only one task. However, the class should not contain too many public operations. A large public interface of a class may indicate that the object is assigned too many responsibilities. Instead the object should probably be split and the responsibilities of the object should be distributed between several objects.

The easiest way to find the operations is probably to look at the MSC messages in the system analysis use cases. The messages can often be considered as operations in the analysis object model. A message received by an instance in an MSC corresponds to an operation on the corresponding class. Note that the operations on the classes representing subsystems define the interface of this subsystem and should often also exist as operations on some of the objects within the subsystem.

Creating an Information Diagram

Now it is time to create an information diagram. This diagram describes the concepts outside the system that the system must know of to fulfill its task. In our Access Control system example, `Card` and `Code` are two such concepts.

1. *Add* a *new* object model diagram to the Organizer in the module `AnalysisObjectModel`. Name the diagram **`InformationDiagram`**.
2. *Open* the `LogicalStructure` diagram from the previous activity and the new `InformationDiagram` in the OM Editor.
3. *Select* and *copy* the classes `Card` and `Code` in the `LogicalStructure` diagram and *paste* them *as* classes in the `InformationDiagram`, while automatically creating the implinks.
4. Consider if any or both of the classes should have any attributes.
5. Associate the classes with each other.
6. Save the diagram.

Your diagram should look like in Figure 737 when you are finished.

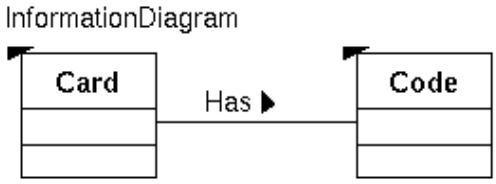


Figure 737: The Information diagram

Creating the Analysis Use Case Model

The design of the two models in the system analysis activity, the object model and the use case model, is usually going on in parallel. The models view the Access Control system from two different perspectives, a dynamic perspective and a static perspective.

The analysis use case model shows the dynamic aspects and consists of a set of MSC diagrams. These diagrams may be categorized into two types:

- Refined requirements use cases
- Behavior pattern use cases

Refined Requirements Use Case

A refined requirement use case is what it reads like. Each valid use case from the requirements use case model is transferred to the analysis use case model and redesigned and refined to the analysis object model.

The purpose of the refined use cases is to validate whether the analysis object model really implements the requirements. At the same time the analysis use case model is an important source of information for identifying operations on the classes in the object model.

In the analysis use case model, the use cases are documented preferably using MSCs. MSC diagrams are more formal and correspond better to the object model than textual use cases.

Each instance in the MSC diagram corresponds to an object or subsystem in the analysis object model. The level of abstraction you choose is a trade-off between detail and clarity.

An object which encapsulates interfaces and just transfers the calls to other objects without adding much functionality may be omitted, but objects providing crucial functionality should be part of the MSC.

Creating a Refined Requirements Use Case

Now you should create an analysis use case of the `Enter_Office_With_Card_And_Code` use case from the requirements analysis activity.

1. Create a new MSC diagram in the `AnalysisUseCaseModel` module and name it **`Enter_Office_With_Card_And_Code_SysA`**.
2. Open the requirements use case `Enter_Office_With_Card_And_Code`. The system analysis use case is based on the requirements use case and it is therefore useful to use the latter as a reference (using copy and paste).
3. Create the MSC on subsystem level, that is, replace the original system instance with instances of `EntranceUnit` and `CentralControl`.
4. Create endpoints of the three instances in the MSC diagram. (Select *Create Endpoint* from the *Link* submenu in the *Tools* menu.)
5. For each message in the requirements use case decide which instances that exchange that particular message in the analysis use case. Also consider which additional messages you need to add. All message exchanges **inside** the system that we did not consider in the requirements use cases have to be added at this point. Also, make references to exceptions at the points where these can occur.
6. Replace the four `ReadDigit` signals with an MSC reference symbol referring to the MSC ReadCode. ReadCode is a behavior pattern which you will create later, see "Behavior Pattern Use Cases" on page 4023.
7. Save the diagram.
8. In the Organizer, create an endpoint out of the newly created MSC diagram. (This is done in the same way as in the editors.)
9. Open the Link Manager and connect this endpoint to the `Enter_Office_With_Card_And_Code` MSC from the requirements use case model. Name the link **`Implementation Link`**.

The created MSC should look like in Figure 738.

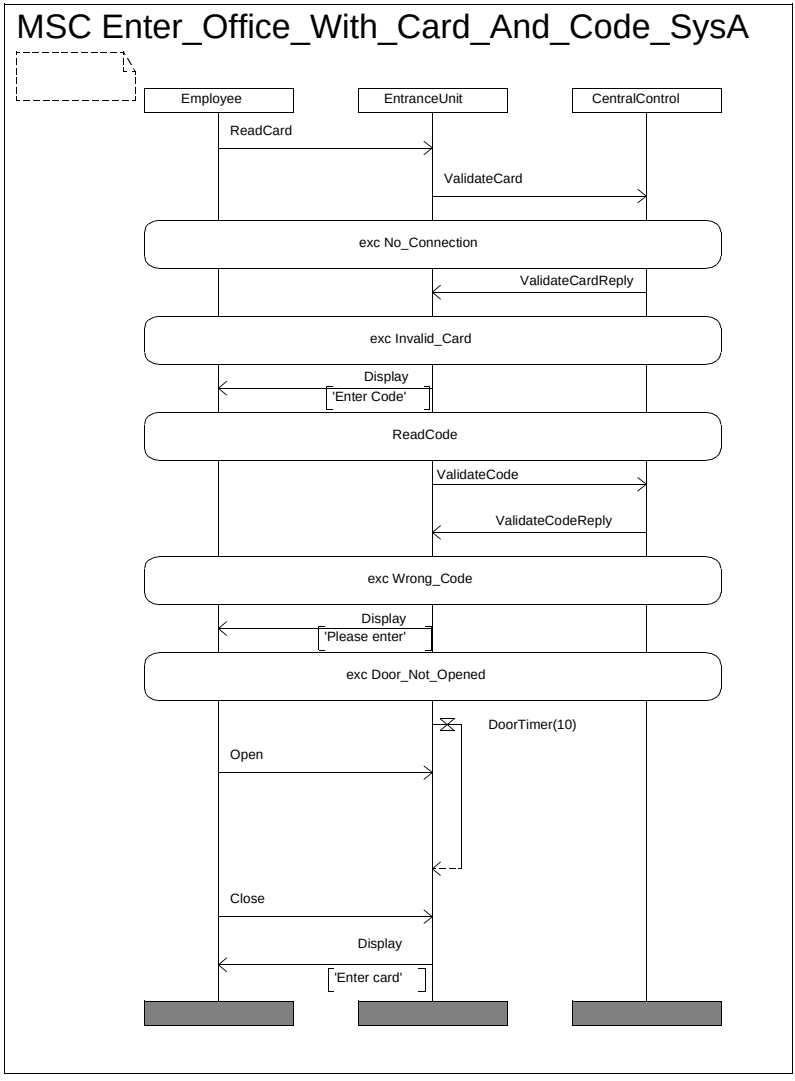


Figure 738: A system analysis MSC use case

The rest of the requirements use cases and their respective exceptions are refined in a similar way. This will not be done in this tutorial because that would take too much time.

Behavior Pattern Use Cases

A behavior pattern is a detailed use case that may be used to examine special communication patterns in detail. A behavior pattern is part of an ordinary use case and most often several use cases share a behavior pattern. A use case may include none or several behavior patterns.

Behavior patterns let refined requirements use cases be presented in a higher abstraction level, making these less complex and easier to understand. By focusing use cases on special parts of the system it is easier to understand and maintain the requirements on the involved objects.

Creating a Behavior Pattern

The refined use case that you just created was created on subsystem level. With the help of behavior patterns we can describe what really happens at a certain point in the use case, i.e. which objects that interact and the messages that they exchange. In our case, where we will create a behavior pattern for the task of reading a code, we have to replace the MSC instance `EntranceUnit` with the MSC instances `KeypadInterface` and `EntranceCtrl`.

1. Create a new Organizer module in the `System Analysis Documents` chapter. Name it **`BehaviorPatterns`**.
2. In the MSC Editor, create the behavior pattern `ReadCode`, see Figure 739.
3. Save the diagram using the name **`Behavior_Pattern_ReadCode`**.
4. In the Organizer, associate the behavior pattern MSC with the use case MSC which it really is a part of. That is, associate it with `Enter_Office_With_Card_And_Code_SysA`.

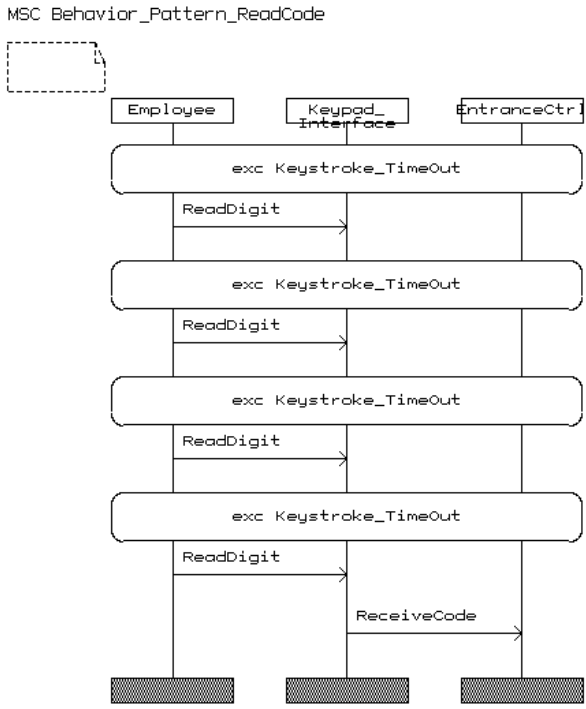


Figure 739: A behavior pattern example

The `System Analysis Documents` chapter should now look like in Figure 740.

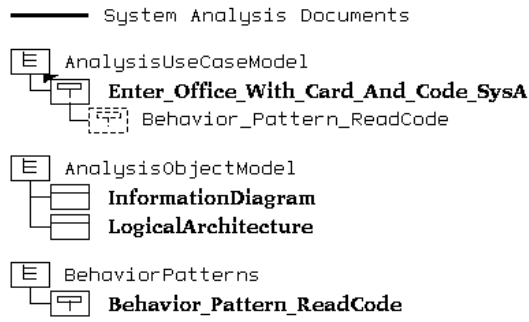


Figure 740: The System Analysis Documents chapter

Requirements Traceability

One important aspect in this activity is the relation between the models created here and the models created in the requirements analysis activity. We want to be able to check:

- That all requirements have been implemented
- Which system analysis object that implements a certain requirement
- Which requirement that is implemented by a certain object in the analysis object model

The means to check the issues above is through consistency checks. There are two types of consistency checks:

- Entity matches
- Link checks

We will start with a link check and then we will perform an entity match.

Link Check

The first thing to check is that all entities described in the logical structure in the requirements object model are either represented in the analysis object model or not really needed by the application.

1. *Open* the Link Manager. A link check can be performed in both entity and endpoint view, so it does not matter which view you have in the window.
2. Choose *Consistency Check* in the *Tools* menu to perform a link check.
3. The Consistency Check dialog pops up asking you to select the documents representing the **from** group. Select the `LogicalStructure` object model and press *Continue*.
4. Yet another Consistency Check dialog appears, now asking you to select the documents representing the **to** group. Select the `AnalysisObjectModel` module (this will also highlight the documents in the module) and press *Check.*
5. The Link Manager will show the result of the link check. You can see that all entities from the requirements, except the system operator, employee, database, management system and office, are represented in the analysis object model. These are concepts on the outside of the system and, thus, not really needed by the application.

To follow links from one model to another we use the *Traverse* command. To see how this works follow the steps below:

6. Go to the `LogicalArchitecture` diagram in the OM Editor and select the class `CentralControl`.
7. In the *Link* submenu in the *Tools* menu, choose *Traverse*.
8. The OM Editor will open the `LogicalStructure` diagram and the `CentralControl` class will be selected. Go yet another step backwards by choosing *Traverse* in this diagram.
9. The Traverse Link dialog pops up asking you to select a link to traverse. The class is the one we just came from so you should choose the text fragment and press *Traverse Link.*
10. The Text Editor opens and the endpoint `centralcontrol` is selected.

In the same way we traversed from system analysis to requirements here, you can also traverse from the requirements to system analysis. Try this!

Entity Match

Now it is time for another consistency check. This time you should check that the instances in the MSC diagram correspond to classes in the object model or to actors that interact with the system.

1. See to it that you have entity view in the Link Manager window. If not, press the *Show endpoint or entities* button in the tool bar.
2. Choose *Consistency Check* in the *Tools* menu. Set the *entity match* radio button in the Consistency Check dialog and press *Continue*.
3. As a document representing the from group, select the MSC `Enter_Office_With_Card_And_Code_SysA`.
4. As documents representing the to group, select the `AnalysisObjectModel` module and the `ActorsList`.
5. The result shows that all MSC instances really are described in the analysis object model or in the list of actors.

Summary

After having completed an entire system analysis, the `System Analysis Document` chapter would look like in Figure 741.

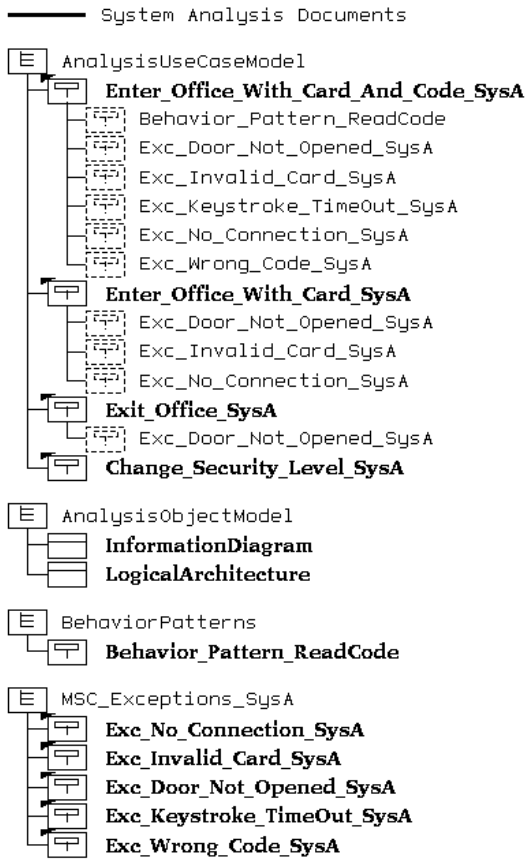


Figure 741: The entire system analysis document structure

Performing the System Design

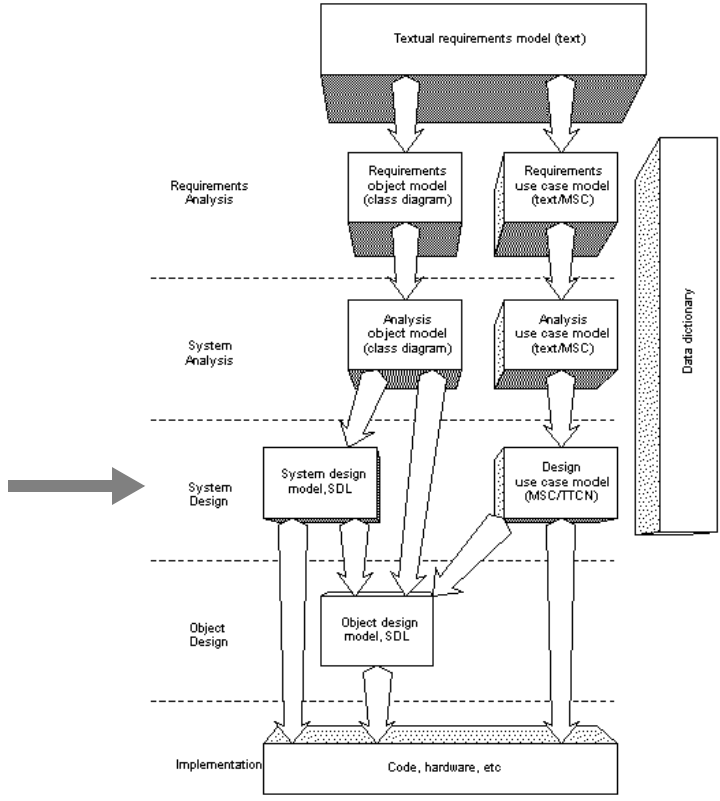


Figure 742: Overview of the SOMT process

What You Will Learn

- To create a design module structure
- To define the static interfaces in packages
- To make an architecture definition of the system
- To make formalized testable use cases
- To use the *Paste As* mechanism when transferring from the system analysis activity to the system design activity

Introduction to the Exercise

This is an exercise on system design. In this activity we no longer make use of object models; from now on SDL will be used. You will learn how to map concepts from the analysis object model from the previous activity into an SDL model. Mapping object-oriented concepts to SDL concepts forces you to make several design decisions. Support for these design decisions is provided through the *Paste As* mechanism. This exercise will teach you to make use of this support.

Useful sources for information in the system design activity are the analysis object model and the analysis use case model. The first provides information about the static structure and is useful when structuring the system into units. The latter provides information about the dynamic structure and is useful for the definition of the interfaces between the units.

Major tasks to perform in the system design are:

- Define the design module structure.
- Define the static interfaces.
- Create an SDL system structure as a starting point for the formalization of the architecture.
- Define the dynamic aspects of the interfaces by a continued use of use cases.

Producing a complete system design structure in the tutorial takes too much time. Thus, you will only perform parts of every step necessary to produce a system design structure.

Preparing the Exercise

1. Open the system file `somttutorial/SysD/accesscontrol.sdt` **(on UNIX)**, or `somttutorial\sysd\accesscontrol.sdt` **(in Windows)**.
2. Check that the Source directory is set to `somttutorial/SysD/` **(on UNIX)**, or `somttutorial\sysd\` **(in Windows)**.

What you now see in the Organizer window is a complete requirements analysis and system analysis structure.

Design Module Structure

An important part of the system design is to divide the system into units considering division of work, distribution of functionality and physical distribution.

The purpose of the design module structure is to show the actual source code modules the application will be built from. The most important aspect of the module structure is that it forms the basis for dividing the work load on different development teams.

In our Access Control system example we have two different subsystems, EntranceUnit and CentralControl. It might be the case that these subsystems are implemented by two different teams. Therefore, it seems natural to introduce two different modules, each module being described by the concept of a package. The contents of the packages will not be defined until the architecture definition, the task here is only to identify the modules needed.

The notation that will be used in this tutorial to describe the design module structure is the object model instance diagram where the instances represent the different modules.

Creating a Design Module Structure

1. Add a new object model diagram to the Organizer in the module `DesignModuleStructure` and name the document **`DesignModuleStructure`**.

2. In the OM Editor, create a first object instance symbol representing the whole SDL system. Name it using the name **`SDL_System_Access_Control`**.

3. Decide which modules, i.e. packages, the SDL system is to make use of. In this example it might be suitable to introduce two different modules, one for each subsystem. Therefore, create two more object instances and name them **`CentralControlPackage`** and **`EntranceUnitPackage`** respectively.

4. Draw associations from the SDL system module to the two new modules. The associations describe that the SDL system uses these two packages.

5. Create yet another module which will consist of all common types and signals. This package is used by the other two packages and thus

also by the SDL system itself. Name the module **`UtilityTypesPackage`** and draw the associations. Your diagram should now look like in Figure 743.

6. *Create endpoints* out of the three packages. (Endpoints are not created automatically for object instance symbols.)
7. Save the diagram giving it the name **`designmodulestructure.som`**.

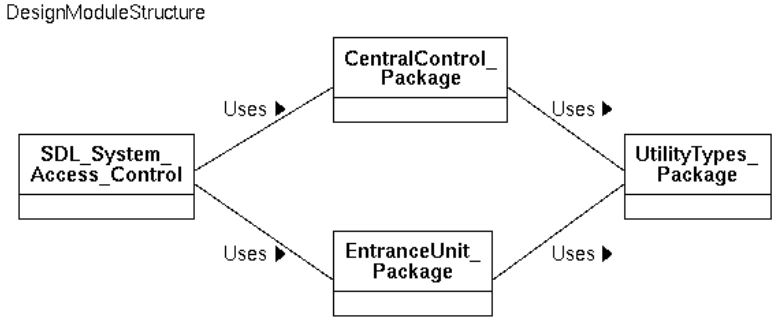


Figure 743: The Design Module Structure

Creating the Architecture Definition

When using SDL to design a system, the architecture is defined by block diagrams which define how the system is decomposed into blocks. The block diagrams are more or less a formalization of the analysis object model.

The first thing we will do when defining the architecture is to define the contents of the packages introduced in the design module structure, see "Defining the Packages" on page 4033. The `UtilityTypesPackage` will contain data type declarations that are common to the subsystems of the system. Signals/remote procedures that make up the interface between the subsystems will also be defined in the `UtilityTypesPackage`. The other two packages, `CentralControlPackage` and `EntranceUnitPackage`, will contain block types and the signals/remote procedures that make up the interface to the particular block. If you have many signals it is often useful to structure these into signal lists.

The basic mechanism used in SOMT when going from analysis to design is the *Paste As* mechanism. This is used here when defining the signal interfaces, the blocks and block types, and, to some extent, the data type declarations.

The second task to do when creating the architecture definition is to define the system by means of SDL block diagrams, see "Creating the SDL System Diagram" on page 4038.

Defining the Packages

Mapping Object Models to SDL Block Types

You should now define the two block types, `CentralControl` and `EntranceUnit` in their respective package using the *Paste As* mechanism.

1. *Add* three *new* SDL packages to the `ArchitectureDefinition` module in the Organizer and name them according to the modules in the design module structure.
2. *Open* the `LogicalArchitecture` diagram in the `AnalysisObjectModel` module.
3. Select the class `CentralControl` and *copy* it.
4. Go to the SDL Editor and the `CentralControlPackage` diagram and choose *Paste As*. The Paste As dialog is opened.
5. *Paste* the class `CentralControl` *as* a *Block Type* (use the option menu) and create an implementation link from the copied object to the pasted object at the same time. The link is automatically created by default.
6. *Copy* and *paste* the class `EntranceUnit` *as* a block type in the `EntranceUnitPackage` in a similar way.

Mapping Object Models to SDL Interface Definitions

Now it is time to design the interfaces of the newly pasted blocks. Interface definitions in SDL are defined using signals and/or remote procedure calls. Consequently, this is what is produced when mapping a class to an SDL interface.

Note that the signals that constitute the interface between our subsystems should be described in the `UtilityTypesPackage`. The rest of the signals, i.e. signals that are **not** exchanged between the subsystems should be described now. Signals from the environment to the subsystem and signals inside the subsystem are such signals.

1. Once again, go to the `LogicalArchitecture` diagram in the `AnalysisObjectModel` and *copy* the class `EntranceUnit`.
2. Go to the SDL Editor and the `EntranceUnitPackage` diagram and *paste* the class *as* a *Text symbol with SDL interface*. See to it that an implementation link is created at the same time.

   If you look at the signal definition you see that the `ChangeSecurityLevel` signal is present. This is a signal exchanged between our two subsystems and, thus, should not be declared here.
3. Delete the `ChangeSecurityLevel` signal.
4. Add the signal parameters. The signals `ReadCard` and `ReadDigit` are the only ones that need parameters. See Figure 744.
5. Save the diagram, giving it the name **`entranceunitpackage.sun`**.

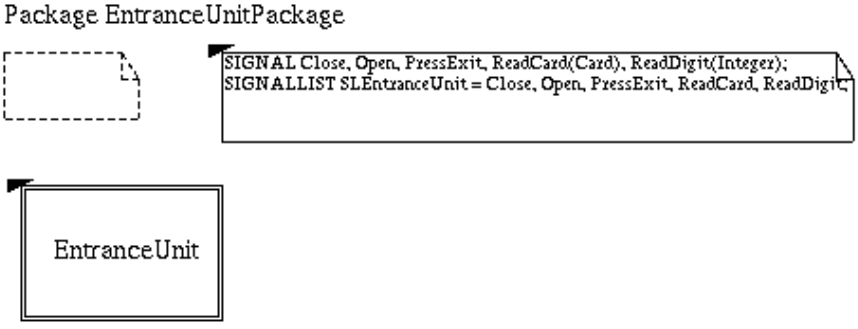


Figure 744: The EntranceUnitPackage

6. Go back to the `LogicalArchitecture` diagram and *copy* the class `CentralControl`.
7. *Paste* it *as* a text symbol with SDL interface in the `CentralControlPackage`.

8. If you look at the signal definition you realize that the `ValidateCard`, `ValidateCode` and `RegisterEntrance` signals are signals exchanged between our subsystems and, thus, should not be defined here. Therefore, delete these signals from the signal definition.
9. Add a signal parameter of the type integer to the `ChangeSystemSecurity` signal, see Figure 745.
10. Save the diagram, giving it the name **`centralcontrolpackage.sun`**.

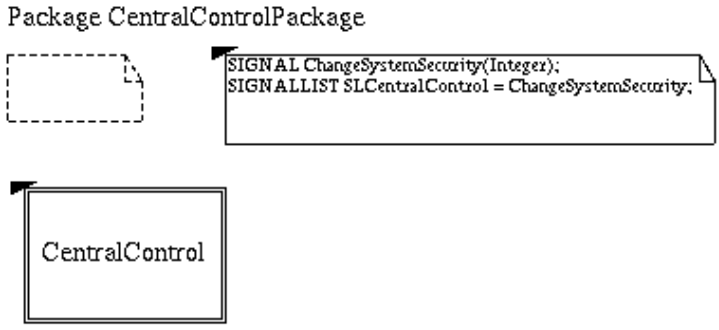


Figure 745: The CentralControlPackage

It is now time to define the last package, the `UtilityTypesPackage`. It will define the common data types needed in the subsystems as well as the interface between the subsystems. To define the package you should use the same procedure as above.

1. *Copy* the class `CentralControl` from the `LogicalArchitecture` diagram.
2. *Paste* it *as* a text symbol with SDL interface in the `UtilityTypesPackage` diagram.
3. Delete the signal `ChangeSystemSecurity` as this has already been defined in the `CentralControlPackage` (because it is a signal from the environment).
4. Add signal parameters to the three remaining signals.

5. Continue with copying the class `EntranceUnit` and *paste* it *as* a text symbol with SDL interface in the `UtilityTypesPackage`.

6. Remove all signals that have already been defined, i.e., all but the `ChangeSecurityLevel` signal.

Note that the Validate commands (`ValidateCard` and `ValidateCode`) have to be implemented with signals and not remote procedure calls. This is due to the fact that we must be able to keep track of how long it takes before we get the answer of a validation (to find out if there is a connection failure or not). As a consequence of this, the reply signals to the Validate commands have to be defined here.

7. Add the two signals `ValidateCardReply` and `ValidateCodeReply` to the latest pasted signal definition.

8. Add signal parameters to the three signals in the definition.

Mapping Object Models to SDL NewTypes

Now it is time to declare the common data types. In our Access Control system example the concepts of card and code are data types common to all parts of the system.

The data type `Card` is best declared as a SYNTYPE. You will have to do this declaration manually as there is no support in the *Paste As* mechanism for pasting something as a SYNTYPE. In declaring the data type `Code`, however, the *Paste As* mechanism can be used.

1. Declare the `Card` concept as a SYNTYPE in the `UtilityTypespackage`, see Figure 746.

2. *Open* the `InformationDiagram` in the `AnalysisObjectModel` and *copy* the class `Code`.

3. Go to the SDL Editor and the `UtilityTypesPackage` and choose *Paste As*. The Paste As dialog is opened.

4. *Paste* the class `Code` *as* a *NEWTYPE with graphical operator* or as *NEWTYPE with textual operator* (it does not matter which one you choose as the class Code has no operator) and see to it that an implementation link is created at the same time. This will only give you the structure, you have to fill in all relevant information yourself.

5. As it is a type we are declaring, change the name `Code` to **`CodeType`**. Define the `CodeType` concept to be an array consisting of four integers (you will have to delete the word STRUCT). The index type of the array should be integer and you will have to define this type too, see Figure 746.

6. Declare a SYNONYM **`NbrOfEntrances`** that will be used to alter the number of entrances we are to control in our system. For now, you can set the value of the variable to **`1`**, see Figure 746.

Before you save and close the diagram you should create an implementation link between the class `Card` from the InformationDiagram to the definition of `Card` in this diagram. You have earlier used the Link Manager to manually create links. We will now show how to use the *entity dictionary* for this.

The entity dictionary is available in all editors and lists all entities in the system, i.e. all modules, diagrams, and endpoints. The main usage of the entity dictionary is to reuse names of entities, but it can also be used to create links.

1. In the `UtilityTypesPackage` diagram, select the text symbol containing the `Card` definition.

2. From the *Window* menu, choose *Entity Dictionary*. The Entity Dictionary window is opened. You will recognize the structure and icons of chapters, modules and diagrams from the Organizer window. In addition, all endpoints are listed beneath the corresponding diagram.

3. Scroll the Entity Dictionary window until you find the class `Card` in the `InformationDiagram` in the `AnalysisObjectModel` module. Select the icon representing the class `Card`. (Make sure not to double-click an icon, since that will copy the name of the icon into the current diagram!)

4. In the Entity dictionary, press the *Create Link* quick button.

5. The Create Link dialog pops up. (This is the same dialog as when you created links in the Link Manager.) Set the *from* radio button, name the link **`Implementation Link`** and press *Create*.

6. *Close* the Entity dictionary by using the *Close* quick button.

7. *Save* the diagram, giving it the name **`utilitytypespackage.sun`**.

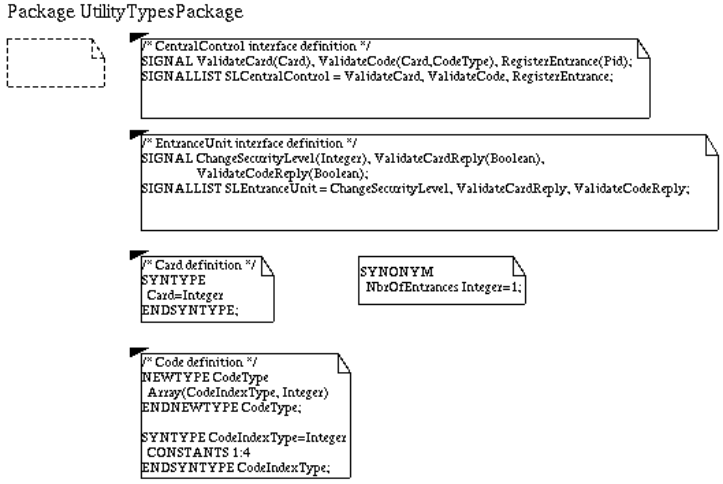


Figure 746: The UtilityTypespackage

8. In the Organizer, *create endpoints* out of the three packages.

Creating the SDL System Diagram

Now you should define the system structure, something which is to be done by means of SDL blocks in a system diagram.

1. *Add* an SDL system diagram to the `ArchitectureDefinition` module in the Organizer and name it **`AccessControl`**.

2. *Copy* the class `CentralControl` in the `LogicalArchitecture` diagram and *paste* it *as* a *Block* in the system diagram. See to it that an implementation link is created at the same time.

3. Also, *copy* and *paste* the class `EntranceUnit` as a *block.*

4. Change the name of the block instances, from `CentralControl` to **`theCentralControl`** and from `EntranceUnit` to **`theEntranceUnit`**. Also, define which block type the blocks are an instance of, see Figure 747.

5. There will often be more than one entrance in a building and therefore you should define the block `theEntranceUnit` as a *block instance set*. (Use the variable `NbrOfEntrances` defined in the `UtilityTypesPackage`.)

The two blocks `theCentralControl` and `theEntranceUnit` must be able to communicate with each other as well as with the environment. The next step is to create all necessary channels in the system diagram.

6. Draw the channels through which the blocks communicate and name them and the gates with suitable names. Also, draw the channels to/from the environment. State which signals that run on a certain channel. To identify the signals consult the `AnalysisObjectModel` made in the system analysis.

7. At this point we see the need to introduce a new signal, the `EnvDisplay` signal which goes from the `EntranceUnit` to the environment (i.e, to the display hardware). This signal contains information that is to be read by persons in the system environment. The parameter of this signal is `Charstring`. Define it in the `EntranceUnitPackage`.

8. Reference the packages the SDL system makes use of in a USE clause in the top of the diagram. Your system diagram should now look like in Figure 747.

9. Save the diagram, giving it the name **`accesscontrol.ssy`**.

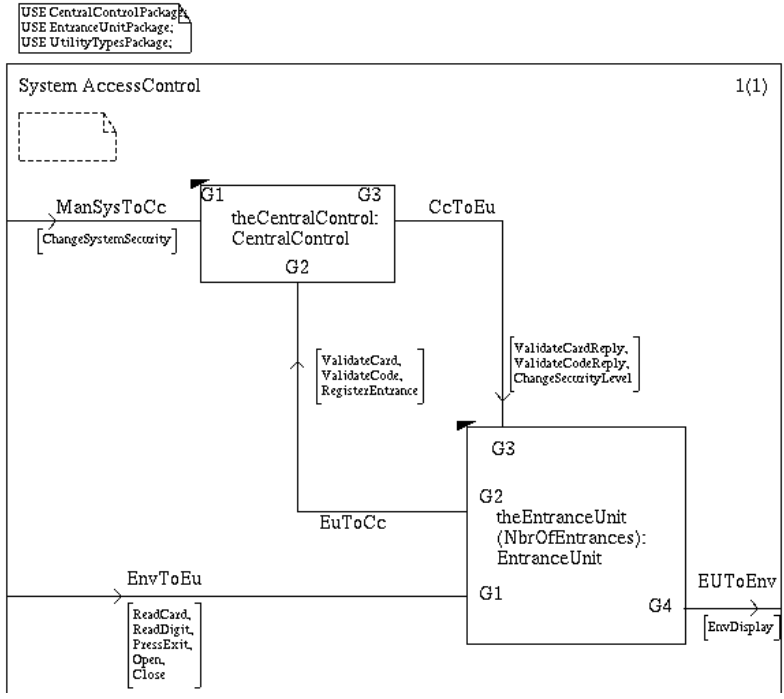


Figure 747: The Access Control system diagram

Refining the EntranceUnit – Mapping Aggregations

Considering the aggregation structure in the `LogicalArchitecture` diagram, the block `EntranceUnit` may be divided into several sub-blocks.

When mapping an aggregation structure from the analysis object model to an SDL diagram, the most common choice is to map the assembly class to a block type. Classes which are part of the assembly class will be mapped to process types, or, if the part class itself is an assembly class, to block types.

The block type corresponding to an assembly class will thus contain processes or blocks.

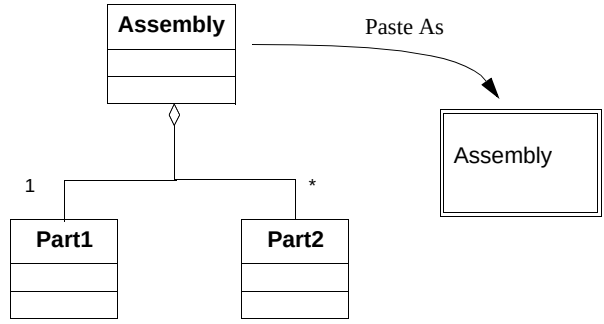


Figure 748: Mapping aggregation to a block type

The process types and block types should be placed in a suitable package to enable reuse. The instances of the types should then be placed in the appropriate system or block diagram to form the system.

Mapping Classes to Block Types and Signal Interfaces

1. Select the class `DoorUnit` in the `LogicalArchitecture` diagram. *Copy* it and *paste* it *as* a block type in the `EntranceUnitPackage`.
2. Call the *Paste As* command again. This time, choose to paste the class as a text symbol with SDL interface in the `EntranceUnitPackage`.
3. The resulting signal definition contains two signals that already have been defined in the package, namely `Open` and `Close`. Therefore, delete these signals in the newly pasted signal definition.

Mapping Classes to Blocks

Now it is time to place a block instance of the `DoorUnit` block type in the `EntranceUnit` block type.

1. Double-click on the `EntranceUnit` block type in the `EntranceUnitPackage` diagram. The Edit dialog pops up, press *OK*. The Add Page dialog pops up. Create a *block interaction page*.
2. For the third time, choose *Paste As* and paste the class `DoorUnit` as a block in the `EntranceUnit` block diagram. The block will be named *DoorUnit*.

3. Change the block name so it is marked as an instance of the block type according to the SDL syntax. Change the name to **`theDoorUnit:DoorUnit`**.

Introducing new Block Types

Since SDL does not allow processes and blocks at the same level you have to create two more block types to be placed in the `EntranceUnitPackage`. These block types will be used as containers to the remaining classes that will be mapped to processes and process types. Give the first block type the name **`EntranceInterface.`** We introduce this block as a generic term for all those classes that represent an interface to the system, i.e. the cardreader, the keypad, the display and the exitbutton. The second block type to be placed in the `EntranceUnitPackage` is **`EntranceCtrl`**.

1. *Copy* the class `DisplayInterface` in the `LogicalArchitecture` diagram.
2. *Paste* it *as* a block type and *as* a text symbol with SDL interface in the `EntranceUnitPackage`.
3. Rename the block type and give it the name `EntranceInterface`.
4. Add a signal parameter called `MessageType` to the `Display` signal definition. By using this type for messages the `EntranceCtrl` does not have to handle strings. This solution makes the system independent of the language used, see "Performing an Iteration" on page 4068 for an example on how this works.
5. Define the NEWTYPE `MessageType`, see Figure 749.
6. Also, *paste* the class `DisplayInterface` *as* a block in the `EntranceUnit` block diagram. Rename the block and give it the name **`theEntranceInterface:EntranceInterface`**.
7. Using the Entity Dictionary, create implementation links between the CardReaderInterface, KeypadInterface and ExitButtonInterface classes in the `LogicalArchitecture` diagram and the `EntranceInterface` block type **and** block.

   Note that you do not have to paste these three classes as SDL interfaces as their operations have already been defined as signals in the `EntranceUnitPackage`.

8. *Now*, *copy* and *paste* the class `EntranceCtrl` as a block type and as a text symbol with SDL interface in the `EntranceUnitPackage`.
9. Delete the signal `ChangeSecurityLevel` as this already has been defined in the `UtilityTypesPackage`.
10. Add signal parameters where they are needed i.e. to the signals `ReceiveCard` and `ReceiveCode`.
11. Also, *paste* the class *a*s a block in the block type `EntranceUnit` diagram. Give the block the name **`theEntranceCtrl:EntranceCtrl`**.

The `EntranceUnitPackage` will look like in Figure 749.

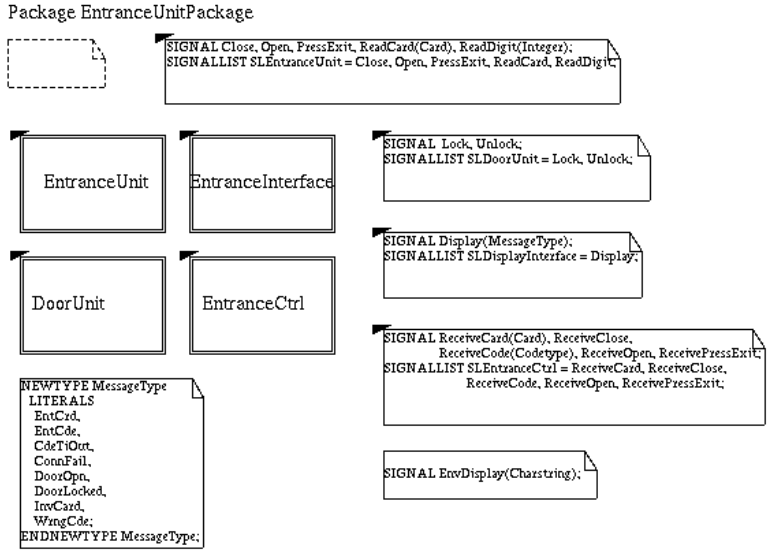


Figure 749: The EntranceUnitPackage

The block type `EntranceUnit` diagram should now contain three blocks: `theDoorUnit`, `theEntranceInterface` and `theEntranceCtrl`.

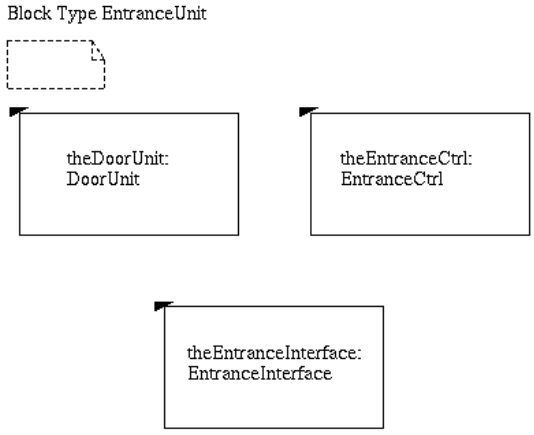


Figure 750: The block Type EntranceUnit

Defining the Communication Structure

Now you should define the communication structures within the block type `EntranceUnit`. Define the channels needed for the three blocks `theEntranceCtrl`, `theEntranceInterface` and `theDoorUnit` to communicate.

To define the channels in the block type `EntranceUnit`, follow the steps described below.

1. Identify necessary channels.
2. Identify the signals that should be carried by the channels. You get a lot of information from the `AnalysisObjectModel` here.
3. Name the channels and the gates.
4. The complete block diagram for `EntranceUnit` should look something like in Figure 751.

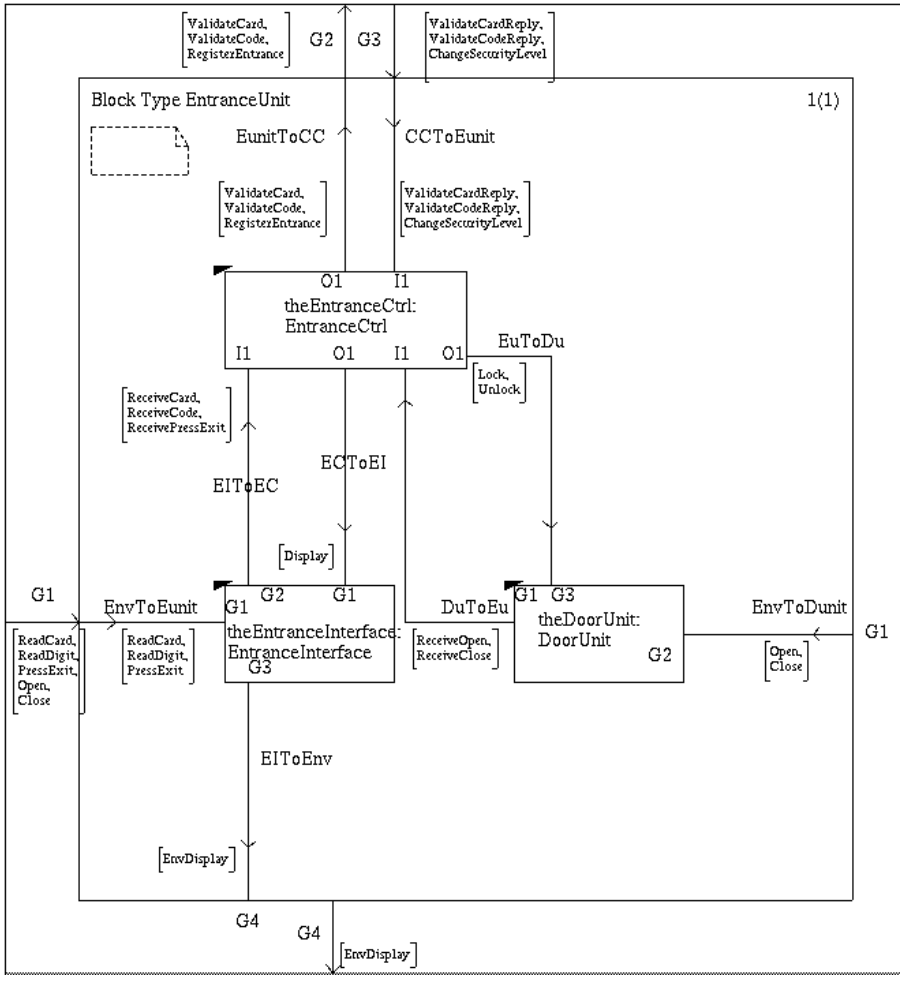


Figure 751: The complete structure of the block Type EntranceUnit

When you have completed the architecture definition, the `System Design Documents` chapter should look like in Figure 752.

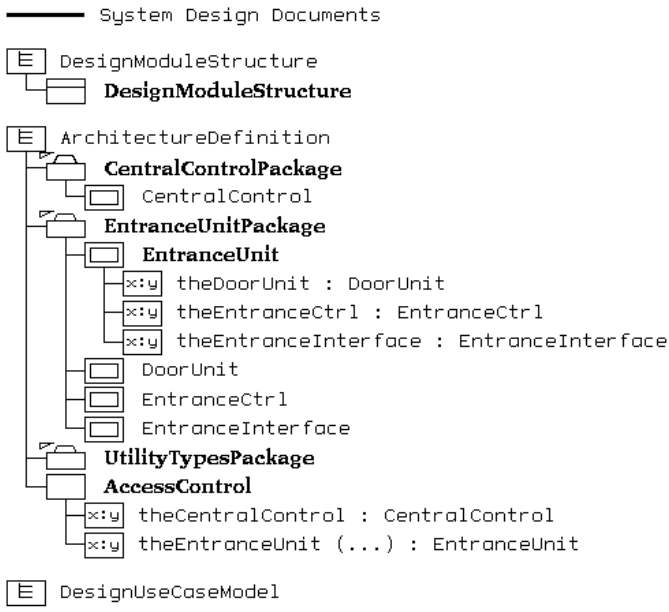


Figure 752: The System Design Documents chapter

Creating the Design Use Case Model

In system design we continue with the use of use cases, this time to define the dynamic interfaces between the blocks in the system.

In our example, the block theEntranceUnit in the system diagram is split into the three blocks, theEntranceCtrl, theEntranceInterface and theDoorUnit. The use cases will consist of the block instances theCentralControl, theEntranceCtrl, theEntranceInterface and theDoorUnit as well as an instance representing the environment.

The use cases must be formalized to a sufficient degree of detail, a level that is consistent with the level of detail found in the static interface definition. Also, the level of detail must be precise enough to make the design use cases act as detailed test specifications.

Formalizing Use Cases

A number of things have to be done when formalizing and refining analysis use cases to design use cases.

- Check that each MSC instance head has a corresponding block in the architecture definition. The use cases define the dynamic interface between the blocks in the system and thus all blocks have to be represented.
- The MSC instances corresponding to actors in the system environment should have **`env_`** stated in their instance head before the actual name, e.g. **`env_Employee`**.
- The MSC instances must have names corresponding to block instances as it is the instances that communicate. That is, the name of those instances that have been defined as block types in the SDL package structure must be changed to the corresponding block instance name. E.g. the MSC instance `CentralControl` can no longer have this name; the name has to be changed to `theCentralControl` according to the architecture definition.
- The messages often have to be replaced with a sequence of message exchanges.
- The MSC messages have to be complemented with parameters according to the definition of signals and remote procedures in the architecture definition. It should not be the **name** of the parameter that should be added but a **value** that the parameter can take. Alternatively the parameters can be skipped entirely.

Creating a Formalized Use Case

Now you should begin to formalize and refine one of the use cases from the system analysis.

1. *Add* a *new* MSC diagram to the `DesignUseCaseModel` module in the Organizer.
2. In the *Add New* dialog, set the toggle button *Copy existing fil*e. As the file to be copied, select the MSC `Enter_Office_With_Card_And_Code_SysA` from the system analysis activity. This file can be found in the `somttutorial/sysanalysis` directory.

3. In the MSC Editor, rename the instance head `Employee` to **`env_Employee`** and change/add the other instance head names. They should conform to the names used on the block instances in the SDL system diagram and the blocks in the `EntranceUnit` block diagram in the architecture definition.

4. For each message, decide if it has to be exchanged with a sequence of messages. If so, add these new messages to the MSC.

5. Add parameters to the messages. Look at the definition of the signals in the architecture definition and add parameters to those messages that are defined to have parameters. E.g. the message `ReadCard` should have a parameter consisting of the card. Note that it is not the parameter name `Card` that is to be used here but an example of a value that the parameter can take, e.g. **`123`**.

6. Rename the use case and save it under its new name, `Enter_Office_With_Card_And_Code_SysD` (use *Save As*).

The exceptions and behavior patterns to this use case must also be formalized.

The other design use cases are created in a similar way. All design use cases should also be connected with implementation links to the corresponding analysis use case and exception. This is done in order to make it possible to check that all use cases from the requirements analysis and system analysis have been refined to design use cases.

Consistency Checks

Now the time has come to performing consistency checks on the models created in system design.

Entity Match

The first thing to check is that the actual modules (SDL packages) used in the design are consistent with the design module structure.

1. In the Link Manager, choose *Consistency Check* in the *Tools* menu and set the *entity match* radio button. Note that you have to be in **entity mode** to be able to perform an entity match.

2. Let the packages in the `ArchitectureDefinition` form the **from** group and the `DesignModuleStructure` module the **to** group.

The result shows that all packages in the `ArchitectureDefinition` really are described in the `DesignModuleStructure`. The consistency view (i.e. the resulting Link Manager window) also shows the contents of the packages. As you can see there are no matching entities to the block references, block types, etc. in the `DesignModuleStructure`. There should not be any, so just ignore this.

Link Check

By doing a link check we will first check that all objects in the analysis object model are mapped to the architecture definition.

1. Once more, choose *Consistency Check* in the *Tools* menu and perform a link check. It does not matter which view you have in the link Manager window, a link check can be performed in either view.

2. Let the `AnalysisObjectModel` form the **from** group and the `ArchitectureDefinition` module form the **to** group.

The result shows that most of the objects from the analysis object model are described as block types and block references in the architecture definition. Many of the objects have also been mapped to an interface definition. The fact that some classes have no corresponding mapping indicates that these classes probably should reside as processes inside some of the mapped blocks. This holds e.g. for the `DoorSensorInterface` and the `DoorLockInterface` as well as for the `SecurityLevel` classes.

At this point you can also select any block or block type in the architecture definition and choose *Traverse link*. The corresponding class in the analysis object model will then be selected. By choosing *Traverse link* again you can follow a link all the way back to the textual requirements. It is also possible to follow a link in the other direction, i.e. from the textual requirements via the object models, to the design. Try this!

Summary

After having completed an entire system design the corresponding document structure in the Organizer would look like in Figure 753.

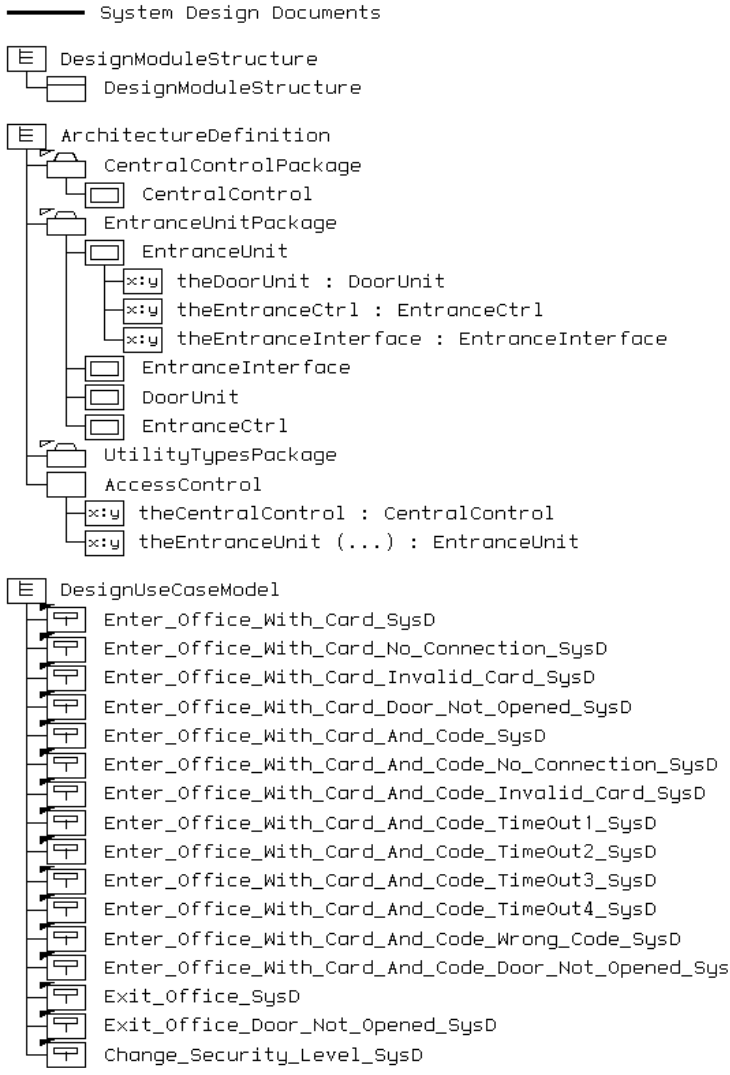


Figure 753: The entire System Design Documents structure

Performing the Object Design

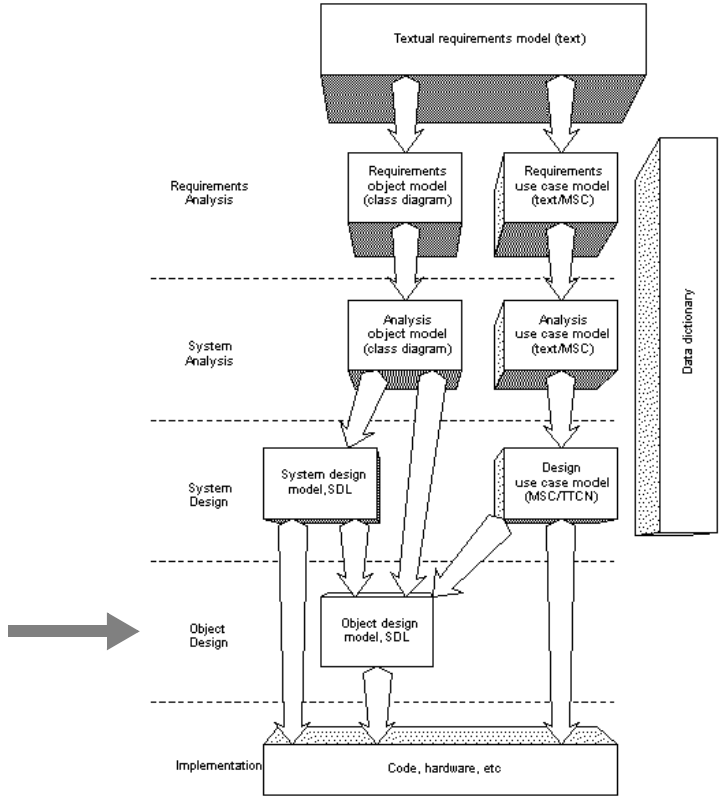


Figure 754: Overview of the SOMT process

What You Will Learn

- To transfer the analysis object model into a consistent object design model in SDL
- To use the *Paste As* functionality to assist the task
- To perform design level testing

Introduction to the Exercise

In this exercise you will perform the object design activity. This activity is, like the system design activity, focused on SDL. However, while the system design is focused on how to structure the architecture and how to decompose the system into blocks, the object design is focused on decomposing the blocks into processes and defining the behavior of the single processes.

The object design activity may be divided into three separate tasks:

1. Map the classes and associations in the analysis object model to suitable SDL concepts.
2. Choose a set of essential use cases and define the behavior of the SDL processes and data types that implement these use cases.
3. Elaborate the design by introducing more use cases and refine the SDL design to handle also these use cases.

Preparing the Exercise

The input to this activity should be a complete requirements analysis, system analysis and system design structure.

1. Open the system file `somttutorial/ObjD/accesscontrol.sdt` **(on UNIX)**, or `somttutorial\objd\accesscontrol.sdt` **(in Windows)**.
2. Check that the Source directory is set to `somttutorial/ObjD/` **(on UNIX)**, or `somttutorial\objd\` **(in Windows)**.

Mapping Active Objects to SDL

An object with its own behavior is called active object. The opposite is an object which acts as an information container - a passive object. Active objects are, most often, mapped to SDL process types. Active objects may also sometimes, as was the case in the system design activity, be mapped to block types.

The default choice in the *Paste As* mechanism is to paste a class copied from an object model diagram as a process type in an SDL diagram.

The attributes of the copied object will be pasted as variables. The operations will be pasted either as signals or as remote procedures of the

process type. This depends on whether the operations are synchronous or asynchronous.

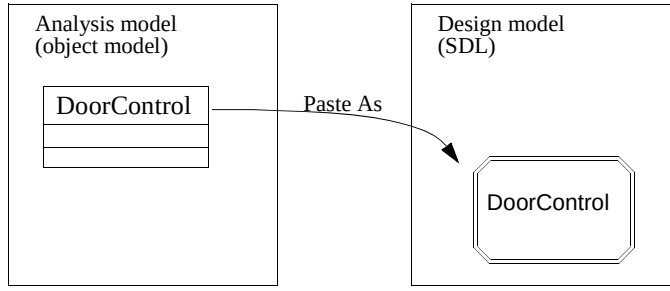


Figure 755: Mapping a class to a process type

Mapping to Process Types

Now it is time to map the classes in the analysis object model that should be mapped to SDL process types. In this example you should map the processes which will reside within the block type EntranceInterface.

The classes which should reside as processes in the block `EntranceInterface` are: the `CardreaderInterface`, the `KeypadInterface`, the `DisplayInterface` and the `ExitButtonInterface`.

1. *Copy* and *paste* each class from the `LogicalArchitecture` diagram as a *Process Type* in the `EntranceUnitPackage`.
2. *Open* the block type `EntranceInterface` diagram in the SDL Editor by double-clicking on the corresponding block type symbol in the `EntranceUnitPackage`. Press *OK* in the Edit dialog. In the Add Page dialog choose to create a *process interaction* page.
3. *Copy* each class once again and *paste* it as a *Process* in the block type `EntranceInterface` diagram.

When pasting e.g. the class `KeypadInterface` as a process, the process will get the same name as the class. This name should be changed since the syntax for process instances requires both an instance name and the name of the corresponding process type. The number of statically and dynamically created instances must also be stated.

4. Name the process **`theKeypadInterface`**. The process `theKeyPadInterface` should have one statically created instance and it should not be possible to create any instances dynamically. The following text should thus be written in the process name area: **`theKeyPadInterface(1,1):KeyPadInterface`**.

   (If the *Remove Reference Symbol* dialog appears, just click *OK*.)

5. Change the other three process names too, according to the above.

Now the block type `EntranceInterface` should contain four processes named: `theCardreaderInterface`, `theKeyPadInterface`, `theExitButtonInterface` and `theDisplayInterface`.

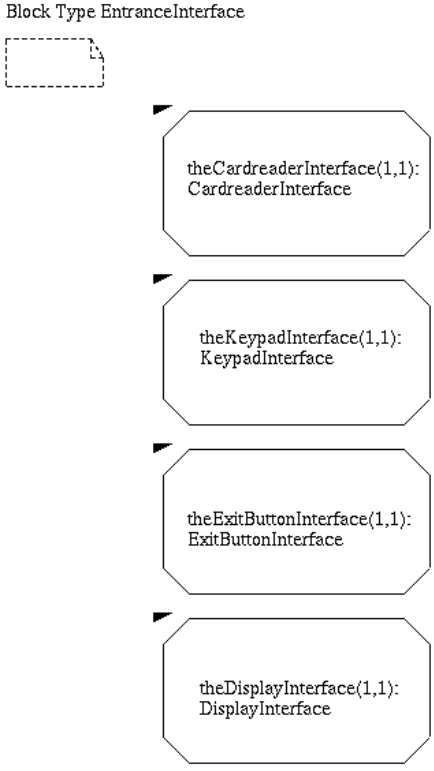


Figure 756: The block type EntranceInterface

Defining the Communication Structure

You have already specified the channels to and from the block `theEntranceInterface` (this was done in the block type `EntranceUnit`). Now you have to connect the processes within this block with the outside.

Creating a communication structure between processes and the border of the block is made in the same way as creating communication structures between blocks, with one difference. The terminology specifies *signal routes* instead of channels as the name for the communication structures at this level. However, there is no practical difference between signal routes and channels.

Now, edit the block type `EntranceInterface` diagram:

1. Connect each process with the border of the block with a signal route in each direction.
2. Give the gate of the input signal to each process the name *Entry* and the other gate the name *Exit*.
3. Specify the signals that are transported on each signal route. Consult the system analysis object model and the specification of the block `theEntranceUnit` to find all signals you should specify.
4. Connect the signal routes with the channels by specifying the appropriate gate for each signal route.

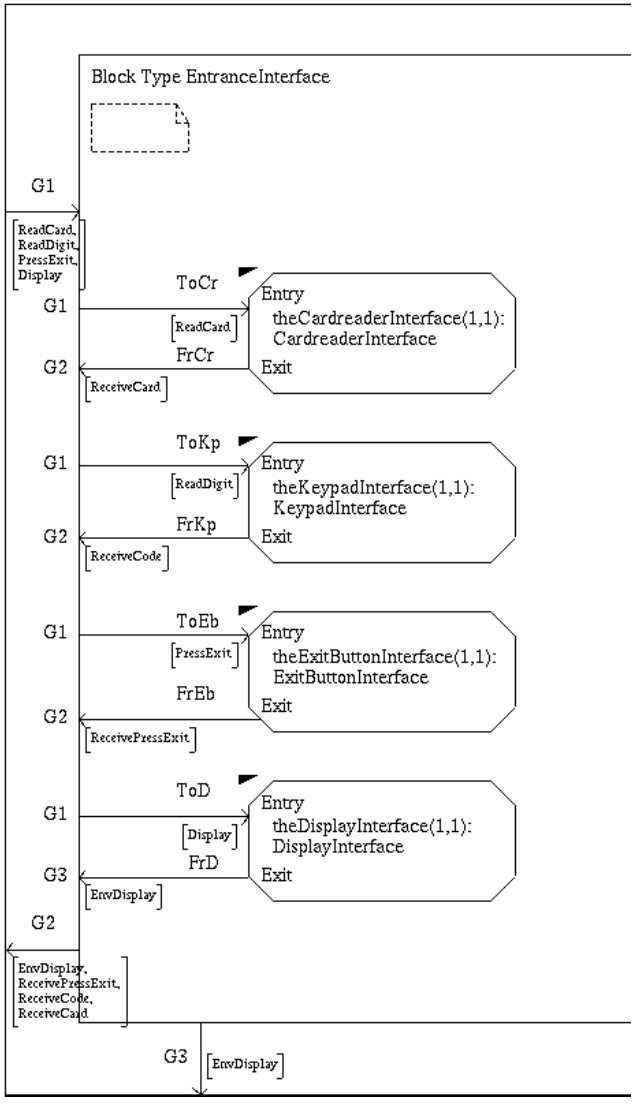


Figure 757: The complete structure of the block type EntranceInterface

Defining the Object Behavior

The activity of describing object behavior is a pure SDL design activity. This section is intended to describe how to structure this activity. It will not focus on specific SDL details.

The most important source of information in this activity are the use cases which specify the signal calls and responses to and from the blocks and processes in the system.

The design of the processes is best made iteratively:

- Select a subset of use cases which describe the most common interaction sequences. Create a basic version of the processes that correspond to these use cases.
- Second, you should introduce a couple of more use cases. Edit the behavior of the processes making them correspond to the extended subset of use cases.
- Continue with introducing more use cases. Edit the processes to cover these, until the behavior of the objects correspond to the complete set of use cases.

Using MSCs to describe the use cases makes it fairly simple to identify the states and transitions of the processes. A transition in a process graph is an input signal, often followed by one or more output signals from the actual process in a use case.

If the situation occurs that two use cases are difficult to combine, you should consider to split the process in question into two separate processes, one process for each use case.

An example is the block type `EntranceInterface` which have a quite complex structure of input and output signals. However, by dividing the block type into four separate processes, each one of these processes becomes fairly simple.

Defining the Basic Behavior of a Process

1. Take a look at the process type `KeypadInterface`. You will see that the process has a signal set of only two signals: the input signal `ReadDigit` and the output signal `ReceiveCode`.

2. If you study the MSC diagram `Enter_Office_With_Card_And_Code_SysD` and the behavior pattern ReadCode you see that four `ReadDigit` signals generate the output signal `ReceiveCode`.

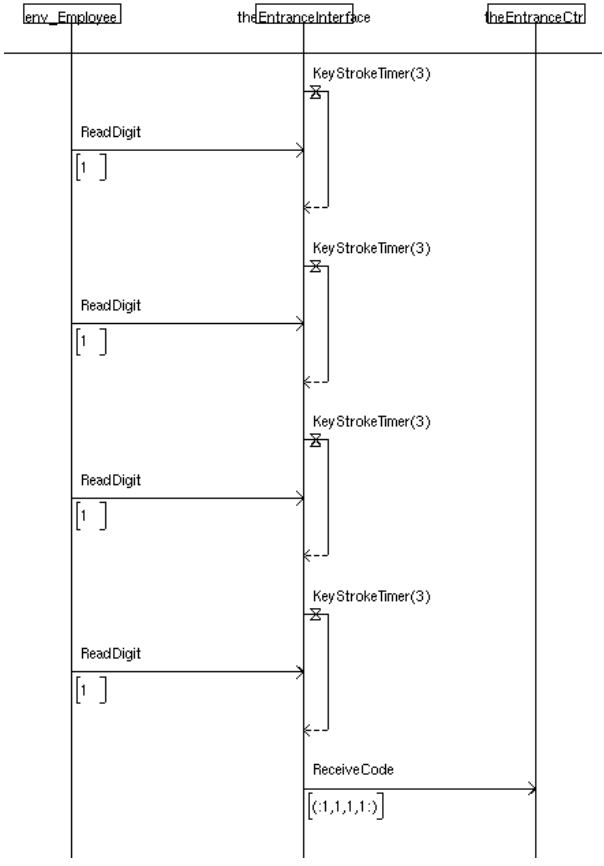


Figure 758: The signal sequence of the process theKeyPadInterface

Creating the behavior of the process out of this information is a quite easy task. Adding the timer, which provides the time-out functionality, will make the first version of the behavior specification of the process complete.

3. In the `EntranceUnitPackage`, double-click on the `KeypadInterface` process type symbol.
4. In the process type diagram, define the basic behavior of the process.

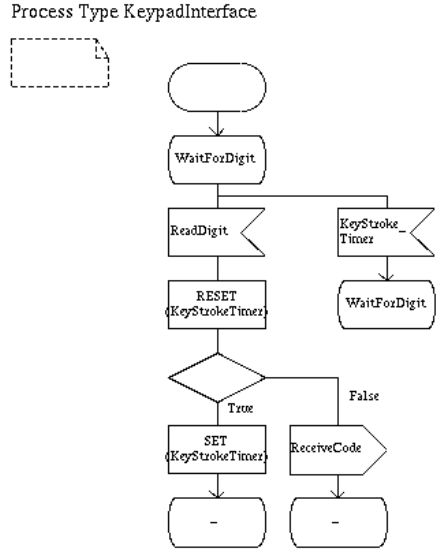


Figure 759: Basic behavior of the KeypadInterface process type

Defining the Data of a Process

The processes will of course also need some data containers, entity variables and control variables. The control variables, such as loop counters and flags, are identified during the object design. Entity variables are most often identified during the system analysis and they may be mapped to the process diagrams from the analysis diagrams.

If we take a look at the process `theKeyPadInterface` again we will notice that we need to introduce a counter to control the number of times a digit will be read before the signal `ReceiveCode` will be sent. We will

also need an index to place the read digit in the correct position of the `Code` array.

1. Place a text symbol in the diagram and declare a `CodeIndexType` named `i` in the process.
2. The parameter of the `ReadDigit` signal is of type integer. Declare a variable named `Digit` of type integer.
3. The parameter of the `ReceiveCode` signal is of type `CodeType`. Therefore, declare a variable `Code` of this type.
4. Change the name of the gate `GKeypadInterface` to `Entry`.
5. Also, add an `Exit` gate and the signal `ReceiveCode` going out of the process type.
6. Declare a timer `KeyStrokeTimer` with duration 3.
7. Refine the behavior of the process.
8. Save the diagram.

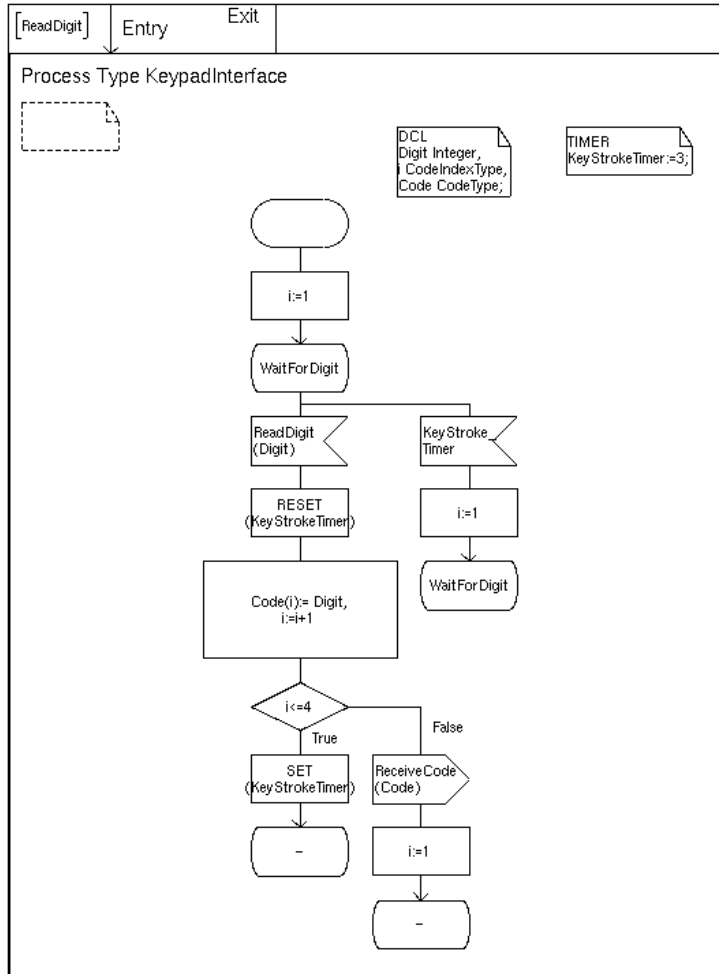


Figure 760: The complete process type KeypadInterface

The rest of the processes in the system are created in a similar way. However, this will not be done in this tutorial.

Design Testing

SDL makes it possible to test the system already during the design. It is possible to simulate an SDL system taking both concurrency and distribution into account. It is also possible to verify requirements specified in MSCs using the Explorer.

The MSC diagrams may, with no or little effort, be used directly as input to the Explorer. This makes the requirements verification simple and efficient.

Preparing the Design Testing

To be able to test your design you must have a complete system.

1. Open the system file
   `somttutorial/objdesign/accesscontrol.sdt` **(on UNIX)**, or
   `somttutorial\objdesign\accesscontrol.sdt` **(in Windows)**.
   What you see in the Organizer now is a complete system structure. All four documentation chapters representing activities from the SOMT method are complete.
2. Check that the Source directory is set to
   `somttutorial/objdesign/` **(on UNIX)**, or
   `somttutorial\objdesign\` **(in Windows)**.

Simulating the System

Simulating the system gives information about how different parts respond to certain inputs and how different parts of the system interacts with other parts. Simulating is often done during the object design to test different parts of the system.

The complete system should also be tested using the simulator to verify that the whole system works as it is intended to.

Try to simulate the system AccessControl. Follow the steps below to make a simulator version of the system AccessControl.

1. Select the system AccessControl in the Organizer.
2. Choose the *Make* option in the *Generate* menu.
3. In the Make dialog, choose the code generator to be *Cbasic*.
4. Choose the standard kernel to be *Simulation*.

5. Press *Full Make*. The system will be automatically analyzed. If there are any warnings you can ignore them. These warnings show up because we have not used all our declared signal lists. As they are just warnings, not errors, you can ignore them.

6. When the make process is finished, start the Simulator by choosing Simulator UI in the *SDL* submenu in the *Tools* menu.

7. In the SDL Simulator UI window, open the file `accesscontrol1_smb.sct` **(on UNIX)**, or `accesscontrol1_smb.exe` **(in Windows)**. Now you can simulate the system, as you have learned in previous tutorials.

Validating the System

When the design of the system is finished you want to verify that the system meets the requirements. This is quite easily done in the SDL Suite by using the SDL Explorer.

MSCs from the system design are used as input to the Explorer.

The requirements use case model is the essential part of the requirements and is often the specification of the system which the customer and the contractor agree upon. Thus, by verifying the system with the MSCs from the system design you are verifying that the system meets the customers requirements.

By making it possible to verify the customers requirements already in the object design and not in a special system design test phase, as in an ordinary design process, you save a lot of effort and time.

Now you should validate some of the MSCs from the system design activity.

1. Select the system AccessControl in the Organizer.

2. Choose the *Make* option in the *Generate* menu.

3. In the Make dialog, choose the code generator to be *Cbasic*.

4. Choose the standard kernel to be *Validation.*

5. When the Make process is finished, start the *Explorer UI* and open the file `accesscontrol1_vlb.val` **(on UNIX)**, or `accesscontrol1_vlb.exe` **(in Windows)**. Now the system is ready to be validated.

6. Press the *Verify MSC* button and choose the system design MSC diagram `enter_office_with_card_sysd.msc` which you can find in the directory `somttutorial/sysdesign`.

7. If the verification succeeds, you will get the message "** MSC <Diagram name> verified **".

It is perhaps too much work to verify all the MSC diagrams of the system design activity during this tutorial and you may quit when you feel that you have understood the principle of how to verify an SDL system.

If all diagrams can be verified against the system, then it is verified that the system also meets the requirements specified in the beginning of this tutorial.

Consistency Checks

There are mainly two consistency checks that should be performed in the object design activity:

- Check that all objects from the analysis object model have been implemented in the design.
- Check that the design model correctly implements the requirements from the design use cases.

The second consistency check is done through design level testing, see "Design Testing" on page 4062. The first one will be performed through a link check, see below.

Link Check

In our complete Access Control system there are implementation links from the system analysis models to both the system design models and the object design models. This implies that we must check our analysis object model against both these design models to see if all the classes have been implemented in the design.

- Perform a link check. Let the analysis object model form the from group. The architecture definition and the SDL design model will form the to groups.

The result shows that all classes but the class `SecurityLevel` have corresponding processes, blocks, signal interfaces or procedures in the design. The `SecurityLevel` does not have any behavior of its own, it is

implemented through its subclasses, and, therefore, the result is just as we want it.

We have now completed the design of the system. It is possible to pick an endpoint in the textual requirements and follow the implink from it, through the object models, to SDL. It is also possible to traverse links the other way, i.e. from design to requirements. Try this!

The use cases are also connected to each other, from requirements to design, as can be seen in the view of the link file in the Link Manager window.

Summary

After having completed an entire object design the corresponding document structure in the Organizer would look like in Figure 761.

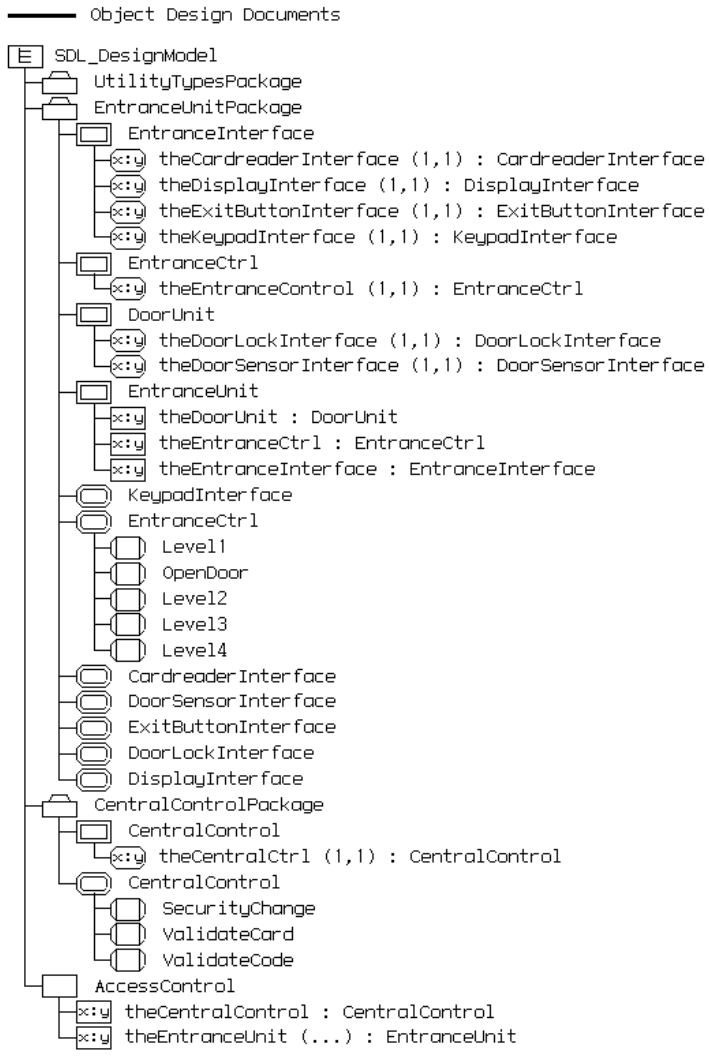


Figure 761: The entire Object Design Documents structure

Implementation

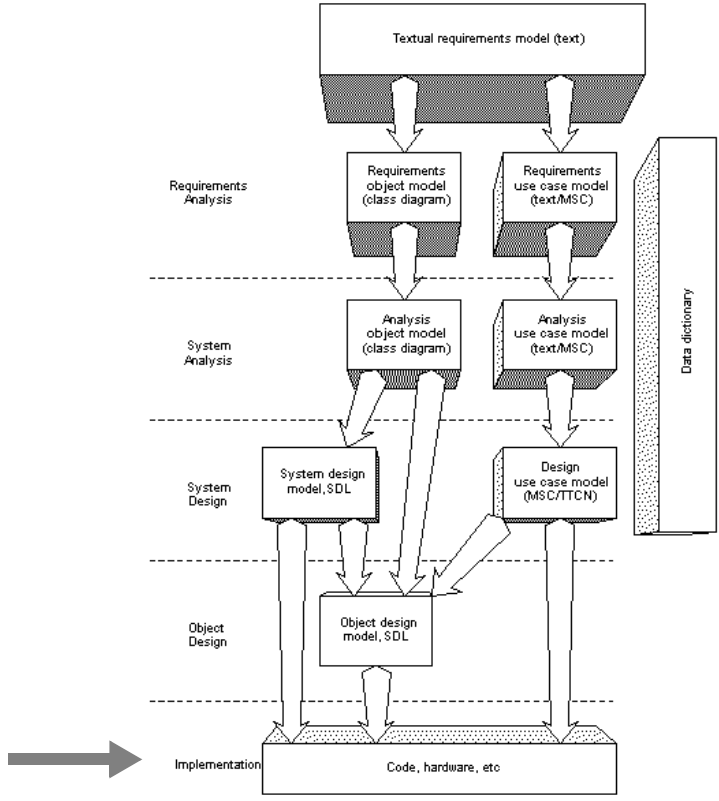


Figure 762: Overview of the SOMT process

The implementation of the system lies outside the scope of this tutorial. For information about the implementation activity, see the SOMT Methodology Guidelines starting in chapter 69 in the User's Manual.

Performing an Iteration

What You Will Learn

- To introduce changes to a system in a controlled way
- To use the *implementation links* to assist you in the activity

Introduction to the Exercise

During the life time of a system new requirements and changes in existing requirements are almost always introduced. We have to be able to handle these requirement changes and adapt the system to the new situation in a controlled way. We call this process iteration. An iteration may also be planned in advance in, for example, incremental development.

This section will describe the scenario of an iteration caused by the introduction of an additional requirement.

Preparing the Exercise

As input to this exercise we will use the complete Access Control system.

1. Open the system file `somttutorial/Iter/accesscontrol.sdt` **(on UNIX)**, or `somttutorial\iter\accesscontrol.sdt` **(in Windows)**.
2. Check that the Source directory is set to `somttutorial/Iter/` **(on UNIX)**, or `somttutorial\iter\` **(in Windows)**.
3. Create a new chapter and name it **`Iteration Documents`**.
4. Add a new module called **`AdditionalTextualRequirements`** to the new chapter.
5. Add the existing file `AdditionalTextualRequirements.txt` to the new module.

Studying the Additional Requirements

- *Open* the additional textual requirements document if it is not already open. The example below shows the document.

Example 656: Additional requirements ––––––––––––––––––––––––

The system should be able handle the languages English, German and French. One version of the system should handle a specific language and the system should be easily configured to handle a new language.

––

As stated in the example above, the task is to redesign the system so it can handle different languages. The design should be made in a way that makes it easy to configure the system to new languages.

The words English, German and French are marked as endpoints in the document.

Examining the Consequences

Now it is time to validate what consequences the new requirement has on the system. The new requirement identify one object that may be affected, the `Display` object.

1. Study the requirements regarding the `Display` in the original textual requirements document. You should find that the new requirement does not contradict with the original requirements.
2. *Traverse* the implementation link from the text fragment `Display` in the original textual requirements. The logical structure diagram in the requirements object model will pop up with the class Display selected.
3. With the class still selected, choose *Traverse Link* once more and follow the link to the logical architecture diagram in the analysis object model. The class `DisplayInterface`, with operation `Display` and attribute `Text` will be selected. The class is a part of the class `EntranceUnit`. It also has a connection to the class `EntranceCtrl`, see Figure 763.

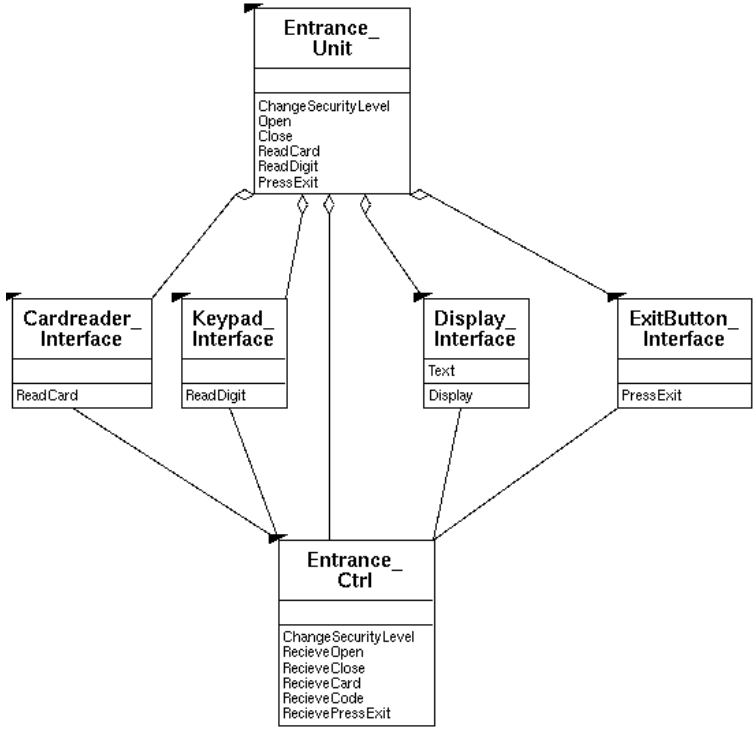


Figure 763: Part of the Logical Architecture diagram

4. Continue to follow the link of the class `DisplayInterface`, from the logical architecture to the package `EntranceUnitPackage`. The class is connected with the block type `EntranceInterface` and the process type `DisplayInterface` as well as with the signal interface defining the signal `Display`.

5. You can also follow the implementation links from the class `DisplayInterface` to the block `theEntranceInterface` (within the block type `EntranceUnit`) and to the process `theDisplayInterface` (within the block type `EntranceInterface`).

6. *Open* the block type `EntranceInterface` in the `Object Design Documents` structure. Study the signal routes leading to and from the process `theDisplayInterface`. You will notice two signals:

`Display` and `EnvDisplay`. Examining the two signals with help of the Signal Dictionary will give you information about the signals. `EnvDisplay` has the parameter `Charstring` and the signal `Display` has the parameter `MessageType`.

7. *Open* the process type `DisplayInterface`. Study the behavior. You will notice that it is possible to change the content of the `EnvDisplay` signal without affecting the behavior of the process type. It also seems like the required changes to the system are limited to the process type `DisplayInterface`.

Introducing Changes in Documents

Now you should introduce the necessary changes to the system to get the desired behavior. Changes should be made in a controlled way. All documents affected of the changes should be edited, to keep the consistency.

Updating the Data Dictionary

1. *Open* the data dictionary. Include information presented in the additional requirements to the data dictionary.

2. *Save* the file as `DataDictionary.txt` in the current directory.

Updating the Requirements Object Model

As we saw earlier, the change we have to introduce to the system is focused on the process type `DisplayInterface`. The new requirement specify that one version of the system should handle a specific language and the only language dependent part of the system is the process type DisplayInterface.

It seems natural to introduce a class for each one of the languages English, German and French in the logical structure diagram. These classes should be subclasses to the class `Display`, i.e. an inheritance structure is needed.

1. *Add* the three new classes to the logical structure diagram in the requirements object model. Name the classes **`FrenchDisplay`**, **`EnglishDisplay`** and **`GermanDisplay`** respectively.

2. *Link* the classes with the corresponding text in the additional textual requirements document.

3. *Save* the diagram *as* `logicalstructure.som` in the current directory.

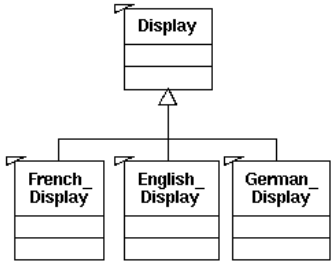


Figure 764: The class Display and its subclasses

4. Make a link from the class `Display` to the class `DisplayInterface` in the logical architecture diagram in the analysis object model.

Updating the Analysis Object Model

In the logical architecture diagram we have to create an inheritance hierarchy corresponding to the inheritance hierarchy in the logical structure.

1. Create the three subclasses to the class `DisplayInterface`. Name the classes **`EnglishDisplayInterface`**, **`GermanDisplayInterface`** and **`FrenchDisplayInterface`**.

2. *Create* implementation *links* to the corresponding classes in the logical structure diagram. (An alternative here would have been to *copy* the classes from the logical structure diagram and *paste* them *as* classes in the logical architecture diagram. The implementation links would then have been created automatically.)

3. *Traverse* the *link* from the class `DisplayInterface` to the process type `DisplayInterface` in the `EntranceUnitPackage`.

Updating the SDL Design

Now it is time to make some changes to the SDL design, making it correspond to the logical architecture. The inheritance hierarchy of the `DisplayInterface` classes should be mapped to an inheritance hierarchy of process types.

1. Create two new process types in the `EntranceUnitPackage`. Name the process types **`GermanDisplayInterface`** and **`FrenchDisplayInterface`**. The already existing `DisplayInterface` process type will function as the `EnglishDisplayInterface`.

2. Mark these process types as endpoints and create links between them and the corresponding classes in the logical architecture. (Alternatively, use the Copy-Paste As mechanism.)

   Now, edit the `DisplayInterface` process type to make it more general and suitable for reuse.

3. Double-click on the `DisplayInterface` process type in the `EntranceUnitPackage`.

4. Create a variable of type `charstring` for each possible message that can be sent to the environment. You will need eight such variables.

5. Put the text **`Virtual`** in the start symbol.

6. Insert a task symbol just after the start symbol.

7. In the task symbol you should initialize the message variables to the corresponding message.

8. Edit each output symbol making it use the corresponding message variable as a parameter instead of a text string. The process type `DisplayInterface` will use English language.

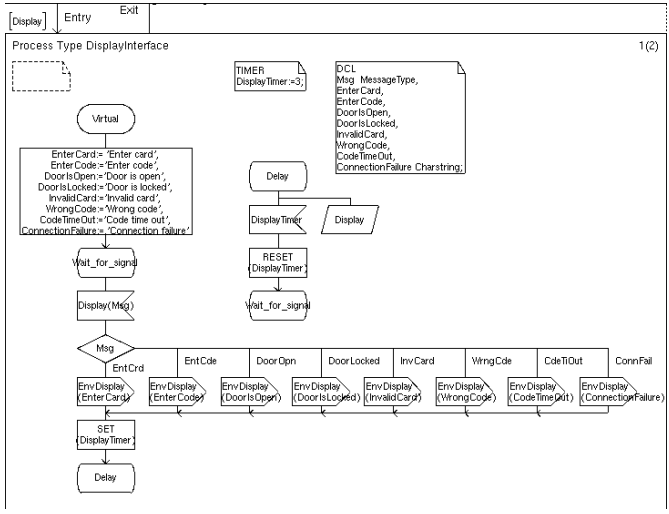


Figure 765: The process type DisplayInterface

9. *Save* the diagram in the current working directory.

After editing the `DisplayInterface` process type it is time to create the process types `GermanDisplayInterface` and the `FrenchDisplayInterface`. For each process follow the steps below:

10. Open a graph page for each process type by double-clicking on the corresponding process type symbol in the `EntranceUnitPackage`. Press *OK* in the edit dialog.

11. In the additional heading you should enter the text:
    `INHERITS DisplayInterface ADDING;`

12. Place a start symbol in the diagram and enter the text **`REDEFINED`** in the symbol.

13. Connect a task symbol to the start symbol. The task is to initialize the message variables to the appropriate messages. Do not use national characters.

14. Connect a state symbol with the task symbol and name it **`Wait_for_signal`**.

15. Save the diagrams.

Now the necessary behavior is described and it is time to show how to configure the system to make use of the new design.

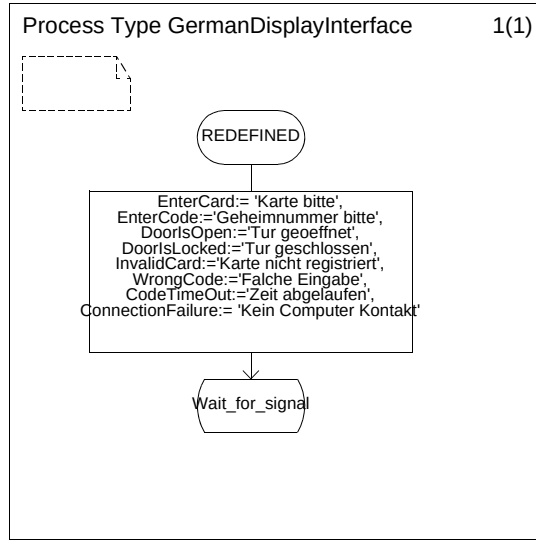


Figure 766: The process type GermanDisplayInterface

Configuring the System

The necessary design is done and we want to configure our system to make use of the new design. Following the steps below will configure the system to a German version.

1. Open the block type `EntranceInterface`.
2. Select the process `theDisplayInterface` and change the text `theDisplayInterface(1,1):DisplayInterface` to **`theDisplayInterface(1,1):GermanDisplayInterface`**
3. Now you can analyze the system. Perhaps you want to simulate the new version of the system; if so, follow the steps described in "Simulating the System" on page 4062.

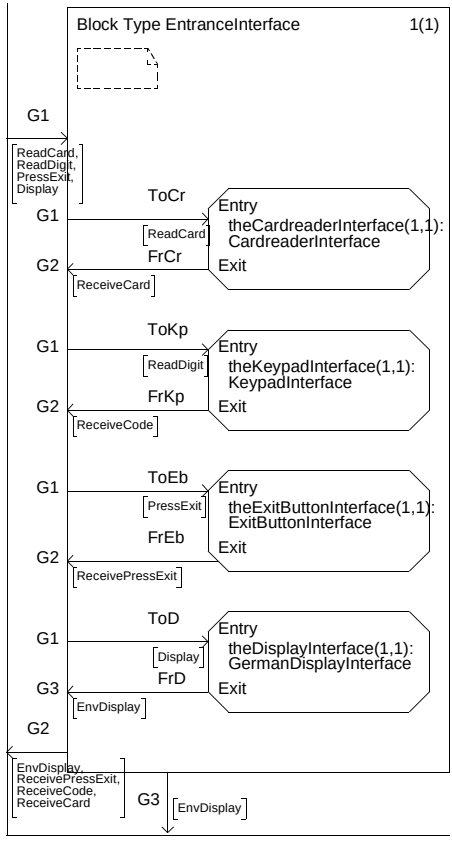


Figure 767: The block type EntranceInterface

Now you have completed the iteration exercise.

To Conclude...

You have now learned the steps of the SOMT method, and we hope you have enjoyed the tour.

Once again we would like to point out that the activities are presented in a sequential order in this tutorial just to simplify the reading. In practise, the task of developing a system using SOMT is a highly iterative process. One activity may start before the preceding activity is completed and the models inside an activity are usually created in parallel.

The SOMT method is intended to support the development process, not to control it. In other words, it is a **proposed** way of working. For your own work, you should not feel that you are locked by SOMT, but pick the parts that suit you best.

Chapter 78 SOMT Tutorial

Symbols

A

B

D

M

Q

R